

I510.78
I29r
no.1161
c.3

10810836

Report No. UIUCDCS-R-84-1161

UILU-ENG 84 1712

AUTOMATIC LAYOUT GENERATION
OF NMOS FIXED FUNCTION CELLS

ILLINOIS STATE LIBRARY

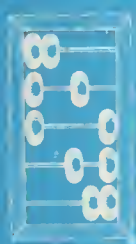
BY

JAMES DONALD SCHWIESOW


MAY 17 1984

April 1984

ILLINOIS DOCUMENTS



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



Digitized by the Internet Archive
in 2018 with funding from
University of Illinois Urbana-Champaign

Report No. UIUCDCS-R-84-1161

AUTOMATIC LAYOUT GENERATION
OF NMOS FIXED FUNCTION CELLS

BY

JAMES DONALD SCHWIESOW

B.S., Valparaiso University, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1984



Urbana, Illinois

April 1984

ILLINOIS STATE LIBRARY

ACKNOWLEDGEMENTS

I would like to thank Professor Kubitz for his advice and guidance in this research and to Mrs. Kubitz for preparing some of the figures. Special thanks go to Joe Luhukay for his assistance and direction when I was attempting to decipher the data structure of the system. In addition, I would like to thank Phil Kos, Jeff Esakov, Paul Yip, and Meng-lin Yu for their assistance throughout this research.

TABLE OF CONTENTS

1.	INTRODUCTION	1
1.1.	Cell Synthesis Methodology	1
1.2.	Cell Requirements For Special Structures	3
1.3.	Overview	5
2.	SOFTWARE ENVIRONMENT	6
2.1.	ICE	6
2.2.	PPF	9
2.3.	Program Use	11
3.	SPECIAL STRUCTURES	17
3.1.	Reserved Area	17
3.2.	Data Structure	19
3.3.	Graphic Representation	22
3.4.	Parameters	22
4.	SPECIAL STRUCTURE DESIGN	23
4.1.	Section I/O Placement	25
4.2.	Parameters	27
5.	SOFTWARE MODIFICATIONS	34
5.1.	Data Structure Modifications	34
5.2.	ICE Modifications	35
5.3.	Geolay	45
5.4.	Symbolay	52
5.5.	TCELL Modifications	53
5.6.	Summary	55
6.	CONCLUSION	56
	APPENDIX A MUX PROCEDURES	60
	APPENDIX B SECTION NODES	67
	REFERENCES	69

CHAPTER 1

INTRODUCTION

Many computer-aided design (CAD) and automated design techniques are presently being developed to cope with the increased complexity of integrated circuit (IC) chips. This increased complexity has caused a very sharp increase in design and development time which in turn has caused a disproportionate increase in the manpower required. For example, a 100,000 device chip may take as many as 120-man-years to design and debug [Murog82]. The development of CAD systems has, therefore, become a necessity for the future design of IC chips.

One approach to a CAD system is presently being investigated at the University of Illinois, Computer Science Department. This system, called *ARSENIC* [Gajs82], is an experimental silicon compiler. ARSENIC will be used to translate an instruction set functional description, given in a Pascal-like language, into a full IC layout. A functional description is input and converted into a structural description. The structural description is then decomposed into various levels. At the lower levels the IC can be described as a set of modules (PLAs, multipliers, ...) and cells (random logic, multiplexers, flipflops, ...). A silicon assembler can then translate the structure into its proper symbolic or geometric description. This paper deals primarily with the expansion of the capabilities of the cell layout synthesis system [Luhu83] being developed as part of the silicon assembler.

1.1. Cell Synthesis Methodology

The goal of the cell layout synthesis system is the automatic generation of the geometric description of digital NMOS cells from an input specification [Luhu83]. The objectives of this system include: controlled cell growth with device size changes, delay time and area estimation and optimization, and interactivity.

The input to this system is the Cell Description Language (CDL) first described in [Bilg82]. The input description of a cell can be divided into three sections:

a. *Global Information (CELL)*

This includes information such as the name of the cell, the model number of the cell, and the

allowed dimensions of the the cell (if area constraints are required).

b. *Functionality* (FUNCT)

Each function in the cell is described in this section and is denoted by the reserved word NOR since these functions are presently limited to AND-OR-INVERT gates. However, once the modifications presented in this paper are fully implemented, CDL may be expanded to allow a library of special functions. These functions may include structures such as MUX (multiplexer), JKFF (J-K flipflop), or REG (register).

c. *Interconnect* (TERM)

This section lists the signal networks (input variables and output functions) of the cell. Each statement gives the relative terminal location and cell side for each input or output of the cell. These locations are on any edge of a cell and may terminate within the cell representing an internal connection or may pass through the cell without making an internal connection in the case of global through-the-cell routing.

The cell description can then be compiled, using *Topocell* (*TCELL*), to produce a topological (abstract) representation [Luhu83]. This representation is a set of net lists which describe the cell in terms of product terms, transistors, and metal routing. The topological data is a network of linked lists [Luhu83]. Each product term, transistor, metal bar, and I/O port is represented by a node in the linked list. Due to the dynamic nature of this data structure, modifications may be made to a cell even after it has been compiled from CDL. This allows the use of a cell editor (*ICE*) [Wong82] which may be used to rearrange an automatically generated cell, with connectivity maintenance, or to create and/or edit a handcrafted cell. A handcrafted cell may be designed by implicitly acting on the topological information and, consequently, the cell design. This topological information is used to produce a symbolic (stick) representation for cell arranging or editing as well as the geometric representation of the cell which is suitable for mask generation in a simplified NMOS process. Mask level information required for fabrication can be produced through the creation of a CIF file.

An example of a CDL description of a cell is shown in figure 1.1. The function $f[1]$ represents the function $(c + da + b)'$. The function $f[2]$ represents $(ac + b)'$ and $f[3]$ represents $(ab)'$. In the term sec-

```

cell    sample (1, , , ) ;
funct   f[1] = NOR (c; d.a; b) ;
        f[2] = NOR (a.c; b) ;
        f[3] = NOR (a.b) ;
term    a : (top1) ;
        b : (top2) ;
        c : (left4) ;
        d : (left5) ;
        f[1] : (bottom1) ;
        f[2] : (bottom2) ;
        f[3] : (top4, bottom4) ;
end.

```

Figure 1.1 CDL Description of a Cell

tion the relative I/O position along the cell boundaries it determined. The variable "a" is input to the cell from the first I/O node along the top of the cell (I/O nodes are numbered from left to right and from top to bottom). The output of function f[2] is output from the cell at the second I/O node along the bottom of the cell. A symbolic diagram of this cell is shown in figure 1.2.

1.2. Cell Requirements For Special Structures

One of the primary goals of this layout synthesis system is to minimize the area of the cell. However, due to the limited interconnect allowed by the system, the area of the cells is not as small as that which would be possible if a more complex interconnect scheme were allowed. The automatic layout system allows only horizontal metal (metal-A) interconnect in the horizontal direction. However, the use of polysilicon and vertical metal-A as interconnect may significantly reduce the area of a cell. Therefore, manually designing many commonly used cells whose functionality is fixed (multiplexers, registers, flipflops, ...), using both polysilicon interconnect and vertical metal-A is beneficial in reducing the overall area of an IC. Parameters may then be used to modify the internal device sizes of these "fixed function cells." Parameterization of device sizes allows the designer (or the automation system) to determine the balance between the power consumption and delay time of the cell. These special structures must fit into the existing topological data structure. Thus, special structures must meet the following requirements:

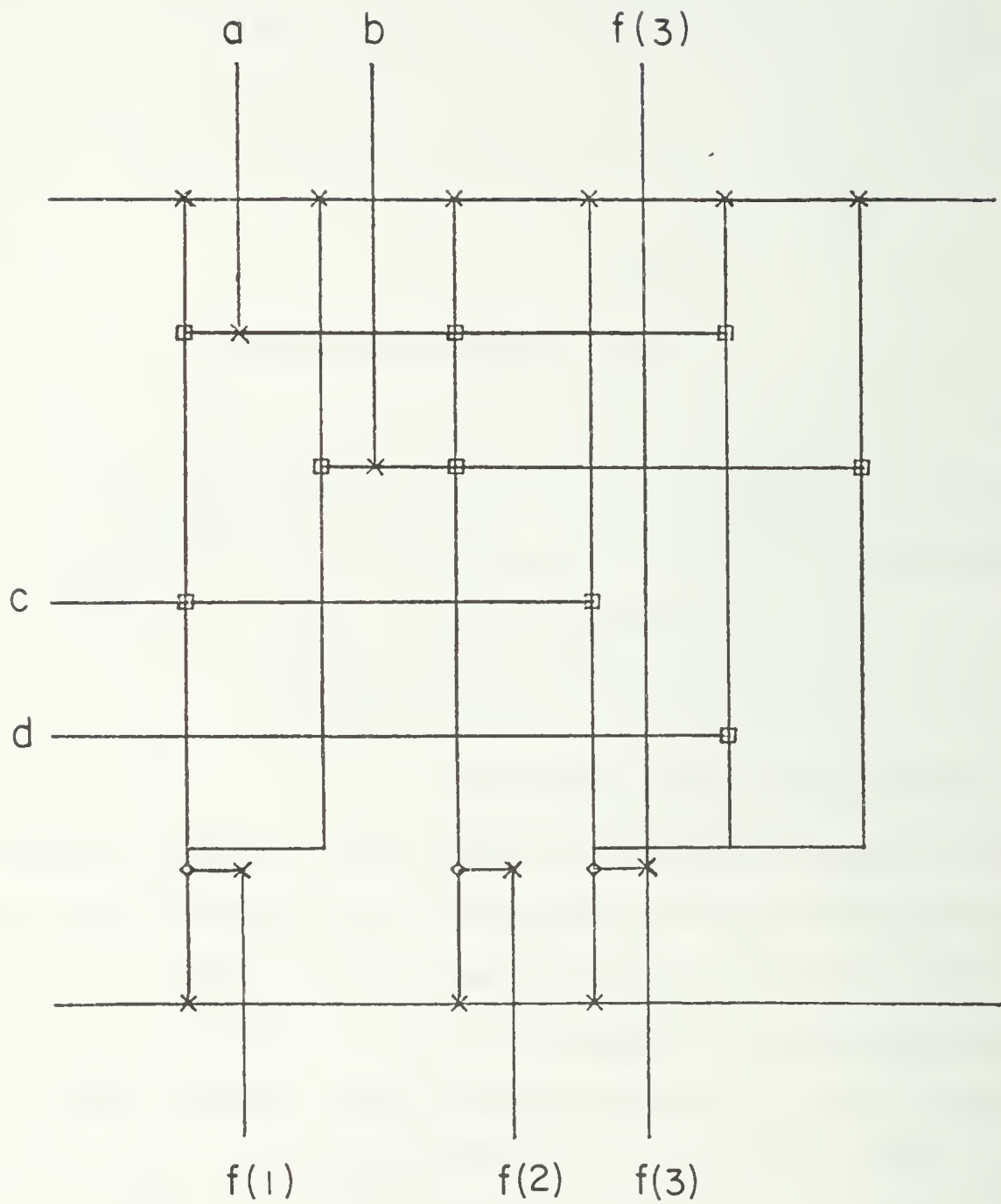


Figure 1.2 Symbolic Diagram of Cell

- Cell Generation:* CDL must be able to input special structures into the net lists of a cell. Without this capability all special structures would have to be manually input into each cell which would defeat the purpose of the automatic synthesis system.
- Editing:* The special structures must be modifiable by ICE. A certain amount of editing, such as the modification of transistor sizes, insertion and deletion of special structures within a cell, and rerouting of I/O and control lines within special structures, must be allowed.
- Connectivity:* Special structures must be capable of abutting with other logic structures. The I/O and control lines must abut correctly with the other structures surrounding the special structure as well as with adjacent cells.
- Intracell Routing:* Special structures must allow routing which completely traverses the structure. This is required when larger cells are developed and special structures are embedded within the cell.

1.3. Overview

This paper describes the modifications which were required to allow the use of special structures in the cell synthesis system. Chapter 2 provides a brief description of the software previously developed for the system along with how to use this software to design cells. Chapter 3 describes the special structures in more detail. This includes a discussion of the sectioning of a special structure. Chapter 4 describes the hard coded layout of a special structure. The software modifications required to add a special structure to the system library, along with an example of a multiplexer structure, are described in chapter 5. Finally, in chapter 6, the results of the present modifications will be discussed along with the possible expansion of the concept of special structures within the system. Appendix A contains a listing of the procedures (macros) used to lay out the sections of a 2-1 multiplexer. Appendix B lists the data declarations which must be added to the data structure for each special structure.

CHAPTER 2

SOFTWARE ENVIRONMENT

The layout synthesis system offers two methods for designing random logic cells. A cell may be designed using CDL, which provides automatic placement and routing, or completely manually designed through the use of ICE. Both of these methods produce cells which conform to the design, placement, routing, and interconnect rules of the system. Special structures, however, do not conform to the set of interconnect rules established by the system (special structure interconnect is discussed in further detail in chapter 3). Therefore, a different method of designing special structures is required.

TCELL and/or ICE create a file which contains the topological data structure of the cell being designed. This topological information is stored as a <cellname>.TPL file. TCELL also creates a file containing the data structure net lists as a set of tables. These tables are helpful for debugging cells or simply reading the topological data structure. These files are in the form <cellname>.LST. The topological information can then be used to produce the layout, or the electrical specifications [Yip84] of the cell. A layout is produced by using the .TPL file as the input file to either GEOLAY or SYMBOLAY. These procedures create the geometric or symbolic representations, respectively.

2.1. ICE

As previously mentioned, ICE is an *Interactive Cell Editor*. ICE can be used to create the topological data structure of a cell or to modify an existing data structure. ICE was developed for this system by Franklin Wong [Wong82]. ICE is accessed through a high level command interpreter *MIDAS* [Esak83]. A block diagram of the cell synthesis process controlled by MIDAS, first diagramed in [Luhu83], is shown in figure 2.1. ICE allows the user to reconfigure a cell at the symbolic level. Each modification at this level implicitly changes the topological data structure of the cell. All reconfigurations then appear in both the symbolic and geometric representation. The memory required to save cells is reduced using this method because the layouts need not be saved. Rather, only the topological data structure must be saved. GEOLAY and SYMBOLAY can then be used to produce the graphic representations on demand.

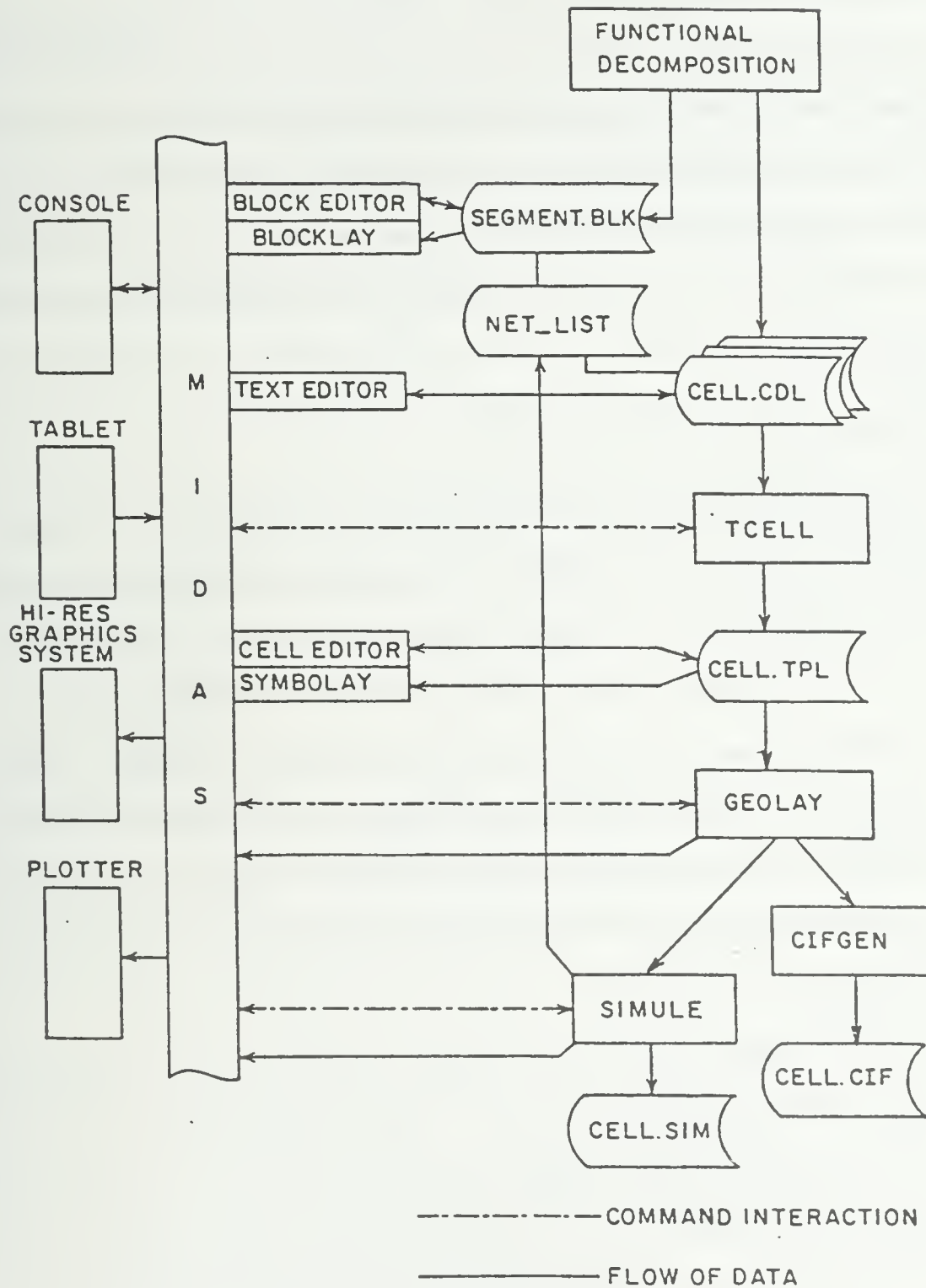


Figure 2.1 The Cell Synthesis Process

GEOLAY and SYMBOLAY are used to produce the graphic representations of a cell regardless of whether the cell was created through TCELL or ICE. Therefore, the topological data structure created by TCELL, for a given cell, must be exactly the same as the data structure created by ICE for an identical cell. This was insured by using the same global variable declarations for TCELL and ICE. These variables are declared in a set of data definition files. These files, along with the files used to produce the layouts and the files which contain special required procedures, make up a set of "include" files. All "include" files are denoted using a *.i* (or in some special cases, *.h*) suffix and are listed in the "constant" block of the main program.

2.1.1. Data Definition Files

The global data is initialized in three data definition files. These files must be included in the ICE and TCELL program files. The method of including the data definition files, along with the remainder of the "include" files, is described later in figure 2.2 on page 12 (*ddeftype3.i*, *geolay3.i*, etc. are simply updated versions of the *.i* files used for special structures). The three data definition files are:

ddefcons.i : This file contains all of the constants which are global to the system. These constants include the maximum number of product terms per function, the maximum number of variables per product term, the maximum number of rows per cell, and the maximum number of variables per cell. Along with these maximums, a set of default values are defined.

ddeftype.i : This file contains the global "type" declarations. The "types" are divided into three sections, the 'function-table' input file, the 'variable-table' input file, and the linked lists. The function-table input file defines *ioname*, *term*, and *funct* which is used when the CDL file is read in by TCELL. The variable-table input file defines the data used by TCELL to create the layout of the cell. The linked lists section defines all of the node types used in the data structure.

ddefvar.i : This file defines all of the global variables. Presently, the global variables are: the *nodefile* (the *.TPL* file), the *headerbase* (pointer to the headers node) [Luhu83], the *total-terms* (the number of product terms in the cell), and *lastterm* (indicates whether the

present product term is the last product term in the cell).

2.1.2. Design Rules File

Another of the "include" files is the design rules file (*desrule.i*). This file is used to store the layout design rules used in GEOLAY to create the geometric layout. These design rules are based upon the layout method developed by Carver Mead and Lynn Conway [MeCo80]. Their design rules use a variable (λ) as a basis for all widths, lengths, distances, and clearances in the layout. The design rules, therefore, are dimensionless. The design rule file contains a list of minimum allowable distances, clearances, and widths in terms of half- λ 's. For example, the minimum width for a metal-A bar (three λ) is defined as: $wmtla = 6$. The use of these λ based design rules adds a great deal of flexibility to the system. The layouts produced by this system are not bound by present fabrication technology. The minimum allowable dimensions used can be updated by simply changing the value of λ , assuming the same linear relationship between all items prevails.

2.1.3. Graphical Procedures

Two graphic representation producing files which must be included in the main program are *geolay.i* and *symbolay.i*. These procedures create graphic representations of the topological data structure. The graphic representations are laid out using a virtual multi-grid system. A graphic representation is produced one column at a time. The rows in each column are laid out from top to bottom. The graphic procedures traverse the linked lists to determine what to draw at the intersection of each row in the column. The layout can then be displayed on a CRT (the Ramtek 9400 color graphics display) or plotted (using the Hewlett Packard 7220T color plotter). The graphics package presently being used to drive these devices is the *Pascal Plotting Facility* (PPF) [MoSC79]. A device independent graphics package (GIGP) [Esak83] is presently being developed to replace PPF in the system.

2.2. PPF

PPF is a Pascal graphics package which was adapted for use on a VAX 11/780 using Berkeley Unix. PPF contains one device independent module (*pbase*) which can be linked to several device dependent modules. The only device dependent modules used are the Ramtek module and the HP plotter module.

The Ramtek device driver source code is in the file *testram.c* and the plotter is in file *hp.c*. The object code of these files must be included, along with the main program, to run on the Ramtek or the plotter (this will be discussed in section 2.3). PPF also requires a set of "include" files declared in the main program (see 2.2.1 below). Further information on the use of PPF in this application can be found in [Wong82].

2.2.1. PPF Global Declarations

As in the case of the data definitions, a set of "include" files must be used with PPF to initiate the global constants, types, and variables. These files are:

- ppfcons2.i*: The PPF constants.
- proctype2.i*: The PPF types.
- ppfvar2.i*: The PPF variables.
- ppfall2.h*: The external procedures and functions used in PPF. These procedures and functions can be classified into two types: external device routines and the Ramtek and plotter library routines. The device routines contain procedures such as maskbox, contacts, loadtransistor, and masklevel. These procedures are called directly by GEOLAY and SYMBOLAY to create the graphic representation of the cell. The library routines contain procedures such as box, line, fill, erase, and readtablet. These procedures are called by the device routines to produce the actual graphic images.

2.2.2. Geolay Macros

In this context a *macro* is defined as a hard coded graphic procedure which may call any number of the device routines in *ppfall2.h*. Several of the device routines (contacts, loadtransistor, drivetransistor) are considered macros because they call other device routines rather than library routines. The procedures used to create the special structures are considered macros because they also call device routines. In addition, a macro procedure may call any number of other specially developed macro procedures. All specially designed macros must be handled as "include" files. This requires that each macro filename contain the *.i* suffix. A specially designed macro, therefore, may be considered to be at a level between the

PPF procedures and layout procedures (GEOLAY and SYMBOLAY). Three macros which have been developed especially to support special structures are:

- drtrans.i* : This procedure produces a geometric drive transistor similar to that produced by *drivetransistor* in *ppfall2.h*. This procedure, however, allows the polysilicon to extend from either the top, bottom, left, or right of the polysilicon to metal-A contact.
- geoldtrans.i* : This procedure is a load transistor which includes a diffusion bar extending above the transistor. The layout is similar to a product term with a load transistor. The only difference is that the diffusion bar does not contain any contact to a metal-A ground bar. The height and width of the diffusion bar is variable along with the load transistor size. This load transistor is used in creating special structures where a ground contact is not desired.
- geomux.i* : This file contains four separate procedures, corresponding to four "sections", which are called by GEOLAY to create the 2x1 multiplexer special structure. These four procedures will be described in more detail in chapter 4 and are listed in Appendix A.

2.3. Program Use

A graphic representation may be created using one of two different methods. The layout may be created using a .TPL file which is run through either ICE, or a program may be written to call PPF procedures directly. Both of these methods require that the main program contain the set of "include" files in addition to GEOLAY and/or SYMBOLAY. How these files are "included" in the main program of ICE is shown in figure 2.2. The file *incldemo.h* is only used by ICE and does not need to be included in other graphic programs. The files *getcell.i* and *savecell.i* are used to retrieve the topological information from a .TPL file and write topological information into a .TPL file (from ICE). The order in which these files are listed is dependent upon their level in the hierarchy. For example, *geomux.i* calls a procedure in *drtrans.i*; therefore, *drtrans.i* must be listed before *geomux.i*. The data definition files are included first because they contain global information. PPF object code files (*ppflib*, *testramlib.o* or *ramlibdemo.o*, and *hp.o*), along with *readdemo.o*, must also be linked into the program. If the plotter is not required, *hp.o* may be omitted. *Readdemo.o* contains procedures used in reading text from the terminal. These files do

```

program icedemo2.p (<parameters>)

#include '/mntb/1/kubitz/schwieso/lfiles/ddefcons3.i'
#include '/mntb/1/kubitz/schwieso/lfiles/ddeftype3.i'
#include '/mntb/1/kubitz/work/ice/src/ddefvar.i'
#include '/mntb/1/kubitz/work/ice/src/desrule.i'
#include '/mntb/1/kubitz/work/ice/src/ppfcons2.i'
#include '/mntb/1/kubitz/work/ice/src/ppfvar2.i'
#include '/mntb/1/kubitz/work/ice/src/proctype2.i'
#include '/mntb/1/kubitz/work/ice/src/savecell.i'
#include '/mntb/1/kubitz/work/ice/src/getcell.i'
#include '/mntb/1/kubitz/work/ice/src/ppfall2.h'
#include '/mntb/1/kubitz/schwieso/lfiles/drtrans.i'
#include '/mntb/1/kubitz/schwieso/lfiles/geoldtrans.i'
#include '/mntb/1/kubitz/schwieso/lfiles/geomux.i'
#include '/mntb/1/kubitz/schwieso/lfiles/geolay3.i'
#include '/mntb/1/kubitz/schwieso/lfiles/symbolay3.i'
#include '/mntb/1/kubitz/demo/ice/incldemo.h'

```

Figure 2.2 ICE "Include" Files

not need to be referenced in the main program. Instead, they are referenced as the program is compiled (Pascal does not allow any referenced procedure to be compiled separately). Therefore, any changes made to PPF or to any of the "include" files requires a complete recompiling of ICE. Manual layout programs, such as special structure macros, do not require any of the ICE procedures while they are being designed. These programs only require the PPF procedures to produce a layout. Therefore, a great deal of compile time and, consequently, design time may be saved during debugging of the macros.

2.3.1. TCELL Use

TCELL is used to create the topological data structure from a CDL input. This data structure must be identical to the data structure used by ICE. TCELL, therefore, must use the same data definition files used by ICE. These files must be "included" in TCELL's main program (*topocell.p*). The file *tcell* is created when *topocell.p* is compiled. An example of how TCELL is run is shown in figure 2.3. If TCELL runs successfully *<cellname>.tp* and *<cellname>.ls* files will be created.

```

>tcell
>>TOPOCELL
      debug mode (y/n) ? y/n
      Inputfile name: <cellname(CDL file)>

      (if no errors occur the following will appear)

      ** CDL scan-in completed 0 errors
      ** Inplace **
      ** Pairs **
      ** Rout **
      ** devsiz **
      > TOPOCELL completed in XXXX msec

```

Figure 2.3 Running TCELL

2.3.2. ICE Use

As previously mentioned, ICE's main program (icedemo2.p) must contain all of the "include" files as shown in figure 2.2. In addition, ICE must also contain the PPF object code when it is compiled. An example of the command line required to compile ICE is:

```

pc -w icedemo2.p /mntb/1/kubitz/demo/ice/ramlibdemo.o
/mntb/1/kubitz/demo/ice/readdemo.o /mntb/1/kubitz/ppf/bin/ppfib

```

The string /mntb/1/kubitz/<directory>/<directory>/<filename> is the path name used in Berkeley Unix to locate a file. The characters pc and -w are a command and option used by Unix [Unix79]. Once compiled, an output file (a.out) is created. Running this file enters the user directly into ICE as opposed to entering ICE through MIDAS. Information concerning compiling ICE into the MIDAS environment may be found in [Esak83]. After a.out is run the user is asked for a device number. The device number is used to declare the Ramtek monitor which is to be used to display the graphic output of ICE (device "1" is normally used). The user is then asked if debug mode is desired. Debug mode produces information concerning the internal procedures. A .TPL (.tp) file is then used as the input file if the user does not wish to manually create a cell. The user may then proceed to use the editing commands through the use

of a tablet connected to the system.

2.3.3. Geometrical Procedures

Manually designed macros may also be run on the Ramtek or on the plotter. To be able to use these devices the macros must be compiled with the proper Ramtek or plotter code files. This may be done through the use of a "makefile" [Unix79]. The format of a "makefile" is as follows:

```

    <macroname>.ram : <macroname>.o /mntb/1/kubitz/ppf/bin/ppfib
/mntb/1/kubitz/demo/ice/ramlibdemo.o
    pc -o <macroname>.ram <macroname>.o
/mntb/1/kubitz/demo/ice/ramlibdemo.o /mntb/1/kubitz/ppf/bin/ppfib

    <macroname>.hp : <macroname>.o /mntb/1/kubitz/ppf/bin/ppfib
/mntb/1/kubitz/demo/ice/hp.o
    pc -o <macroname>.hp <macroname>.o
/mntb/1/kubitz/demo/ice/hp.o /mntb/1/kubitz/ppf/bin/ppfib

    <macroname>.o : <macroname>.p (path names to additional required
files) <macroname>.i
    pc -c -w <macroname>.p

```

The *<macroname>.i* file contains the primary macro procedure(s) and *<macroname>.p* is the main program which calls the procedure(s). Any additional files containing procedures (macros) required by the primary macro must be included preceding the file containing the primary macro. The command: *make <macroname>.ram* or *make <macroname>.hp* will then compile the macro. The *<macroname>.ram* file can then be run to create an image on the Ramtek. The file *<macroname>.hp* will draw the same image on the plotter. As with ICE, the device number (normally "1" for either Ramtek or plotter) must be input along with setting the debug flag. The reference point coordinates must also be input (on both devices, point (0,0) is the lower left corner of the display). The scale factor needed is dependent upon the size of the macro. A scale factor of 2.0 has been found to be a reasonable initial value. After the program procedures are input the layout will be produced.

2.3.4. Plotter

Presently, ICE cannot transfer any code to the plotter, all .tp files to the HP plotter must be through a separate program. Two programs which produce the plotter output are *geoplot* and *symploplot*. These procedures take a .tp input file and use GEOLAY (for *geoplot*) or SYMBOLAY (for *symploplot*) to produce a layout on the plotter. When the device option is set to "1" the layout data is sent directly to the plotter. When the device option is set to "0" a binary output file (*Plotfile.0*) is created. The user then uses the *plothp* program to send the data to the plotter. With the device "0" option *geoplot* calls a compress procedure which eliminates all unnecessary boundary lines in the layout. A compressed metal bar is shown in figure 2.4.



Before compress



After compress

Figure 2.4 Compressed Metal Bar

After the layout is compressed the data is sent to the plotter via sendplotter. A diagram of *geoplot* is given in figure 2.5. *Symploplot* is essentially the same except that SYMBOLAY is used instead of GEOLAY. Figure 1.2 was created using *symploplot*. *Symploplot* uses the device "1" option because compress is not needed in a symbolic layout. Unlike *geoplot*, the initial point (0,0) will reasonably center the plot. *Geoplot* and *symploplot* will use correct mask level colors if the plotter pens are positioned correctly. The pen position along with the color and mask level are shown in table 2.1.

<u>Pen Position</u>	<u>Color</u>	<u>Mask Level</u>
1	Red	Polysilicon
2	Green	Diffusion
3	Blue	Metal-A
4	Yellow	Implant
5	Violet	Metal-B
6	Black	Contact
7	Black	Metal-B Contact

Table 2.1 Color Pen Positions

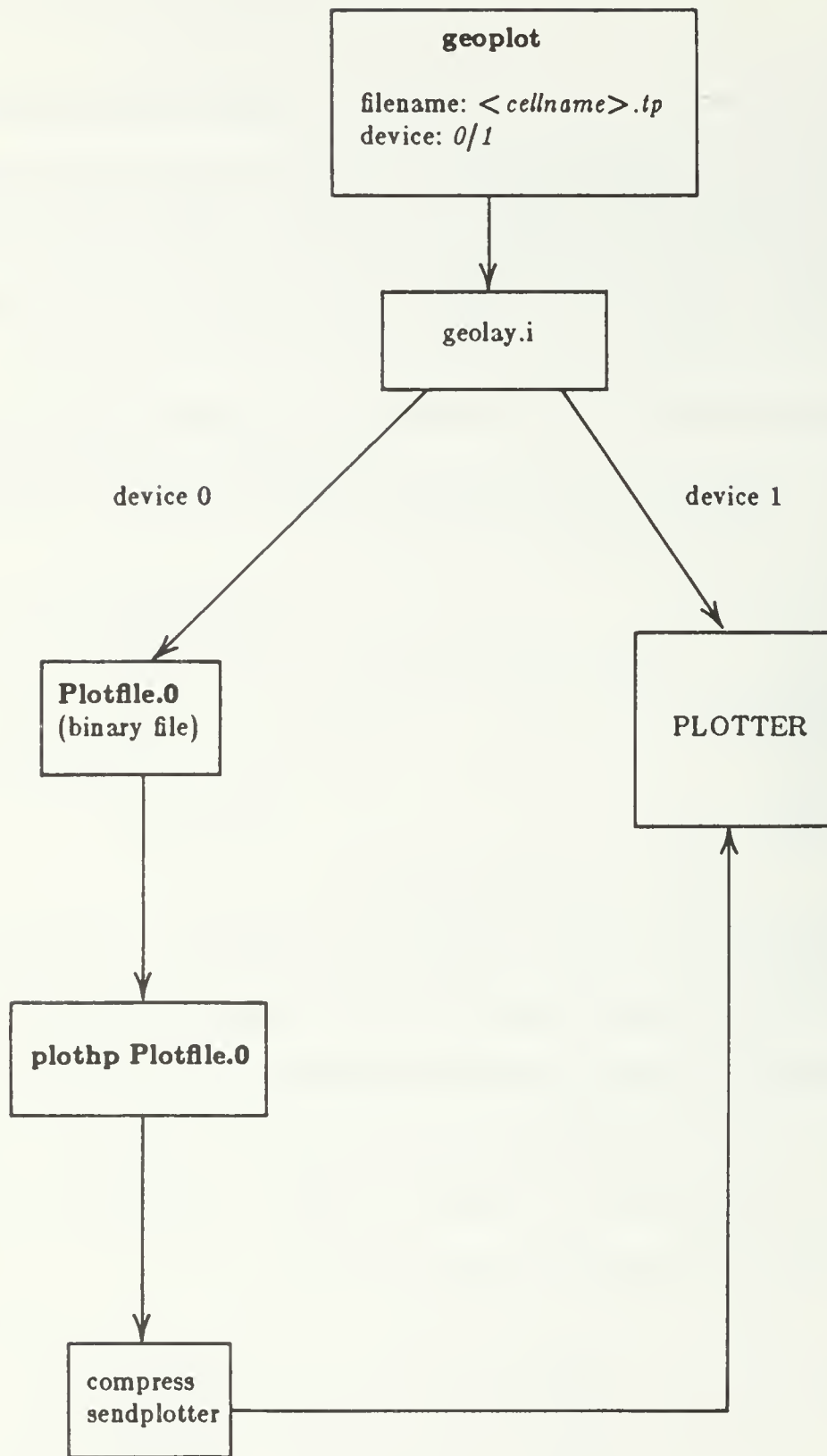


Figure 2.5 Geoplot

CHAPTER 3

SPECIAL STRUCTURES

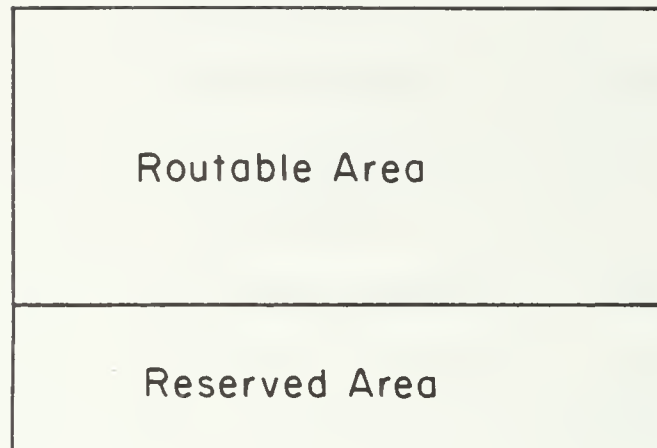
Each special structure is made up of two areas. These areas are the routable area and the reserved area (see figure 3.1(a)). The routable area is treated the same way as normal gatecell structures by the data structure. This allows the routable area to be automatically routed by TCELL and allows editing through ICE. Editing and routing in a reserved area, however, are very restricted. This area is considered reserved for the special interconnect requirements of a special structure. Editing is limited to modifying transistor sizes, insertion and deletion of global metal-B routing, and to changes in the reference point for the special structures. Thus, all routing is limited to the routable area of the structure and to global metal-B bars which completely traverse the reserved area.

3.1. Reserved Area

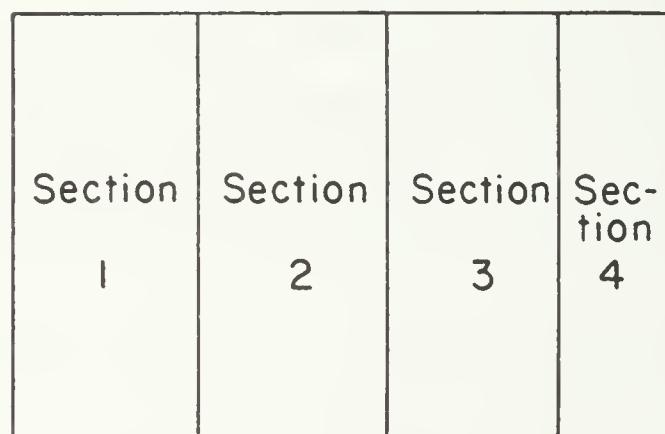
A combination of sections make up the reserved area of a special structure (see figure 3.1 (b)). This area was developed to allow interconnect which is normally not accepted by the present data structure. Polysilicon and vertical metal-A interconnect, for example, are allowed in a reserved area but are not allowed in the normal data structure. This special interconnect requires that the reserved area layout be hard coded. Hard coding the layout allows the designer to manually design the reserved area of the special structure. Manual design also allows the designer to minimize the area of the structure.

As with all cell designs, the reserved area should use the smallest amount of area possible. The primary design objective in this case is to keep the height of the area, rather than the width, as small as possible. This goal can be achieved by placing as many as possible of the horizontal metal-A bars, and their respective drive transistors, above the reserved area (in the routable area). Most control lines (clock inputs, select lines, etc), therefore, are inserted outside of the reserved area in regions where they can be routed by TCELL.

The number of reserved area sections in a special structure are dependent upon the number of product terms required. Each section may contain one and only one load transistor. Each section may also contain, at most, one metal-A node. This node is used to form an I/O contact within the reserved area.



(a) Routable and Reserved Areas



(b) Sectioned Reserved Area

Figure 3.1. Cell With Reserved Area

This restriction occurs because only one set of top-bottom I/O nodes is associated with each section. Consequently, only one metal-B bar may be associated with each section. The metal-A node is required to establish an endpoint for the metal-B bar within the reserved area.

3.2. Data Structure

The data structure of the cells consists of a set of linked lists. These lists include the I/O list, the product term list, the metal-A list, the metal-B list, the drive transistor list, and the load transistor list. Each cell has a header node which points to the beginning and end of each I/O list, the beginning of the metal-A and metal-B lists, and the product term list. Each of these header nodes are linked to (pointed to by) a common node: *headerbase*. Headerbase, therefore, is the node which has access to the entire set of linked lists. For example, the command to access the metal-A list is, *headerbase^.mtlahdr* (for further information, see [Luhu83]).

The I/O linked lists contain information concerning the I/O nodes on each side of the cell. These linked lists are the top, bottom, left, and right I/O lists. Each I/O node can point to either a variable node or can be nil. A variable node is used to identify the I/O bar. The I/O nodes are also used to define the cell boundary. Each node represents an available column, or row, in the cell. In this way a virtual grid system is established by the set of I/O nodes. The vertical and horizontal pitch of the grid, the distance between adjacent nodes, is parameterized so that actual spacing may be determined during the geometry generation phase. In addition to defining the grid, each node may point to a metal node. For example, a left I/O node will point to the first metal-A node in it's row, a right I/O node will point to the last metal-A node. The first metal-B node will be pointed to by the appropriate top I/O node and the last metal-B node by the appropriate bottom I/O node.

The metal-A linked list contains all of the metal-A nodes. Each metal-A node points to the I/O nodes on it's left and right ends. These pointers are used to determine the length of the metal-A node. Metal-A nodes can either begin or end at a drive transistor, an I/O node, or a metal-B node.

The metal-B linked list contains all of the metal-B nodes in the cell. Metal-B nodes have pointers to the top or bottom I/O node and to a metal-A node. Metal-B nodes, therefore, can only connect to metal-A nodes within a cell or to I/O nodes along the upper and/or lower boundary of a cell.

The product term list is a doubly linked list which is accessed through two header links, namely *headerbase^.ptlefthdr* and *headerbase^.ptrighthdr*. Each product term contains a pointer to a top and bottom I/O node along with a pointer to the product term on its left and right sides. Each product term may also contain a drive transistor list and possibly a load transistor or a special structure list. Product terms are grouped into functions and only the leftmost product term of a function may contain a load transistor. Individual functions may have only one load transistor. In the current implementation a function is allowed up to eight product terms, and each product term may have at most three drive transistors. These numbers are implementation dependent and can be changed. An example of the product term list for function *f[1]* in the cell described in figures 1.1 and 1.2 is shown in figure 3.2 (a second header node pointing the list is assumed). A tag (*ptstrtype*) is used to label the structure type associated with each product term (section 5.1.1). In the case of a product term associated with a random logic function, the structure type is set to *nosstr* (no special structure). In the case of a special structure each product term must be labeled with the particular structure identifier. For example, a 2-1 multiplexer is identified as *mux*.

Each product term in a special structure is basically a special case of a random logic product term. Therefore, a separate linked list for special structures is not required. If the product term is identified as part of a special structure, a pointer (*ptss*) from the product term node to the appropriate section node is established. As mentioned previously, each special structure is divided into individual sections. Individual nodes must be created for each section of a special structure. A special structure (section) node must contain a pointer to the load transistor of that section and information pertaining to the size of the drive transistors within the reserved area. If a cell containing a special structure is desired, each successive product term contains a section of the special structure (rather than a random logic node) until the cell is complete. This assures that the structures are always complete and that the sections will abut correctly. Unlike random logic, however, a special structure automatically creates a load transistor, drive transistors, and the required metal routing.

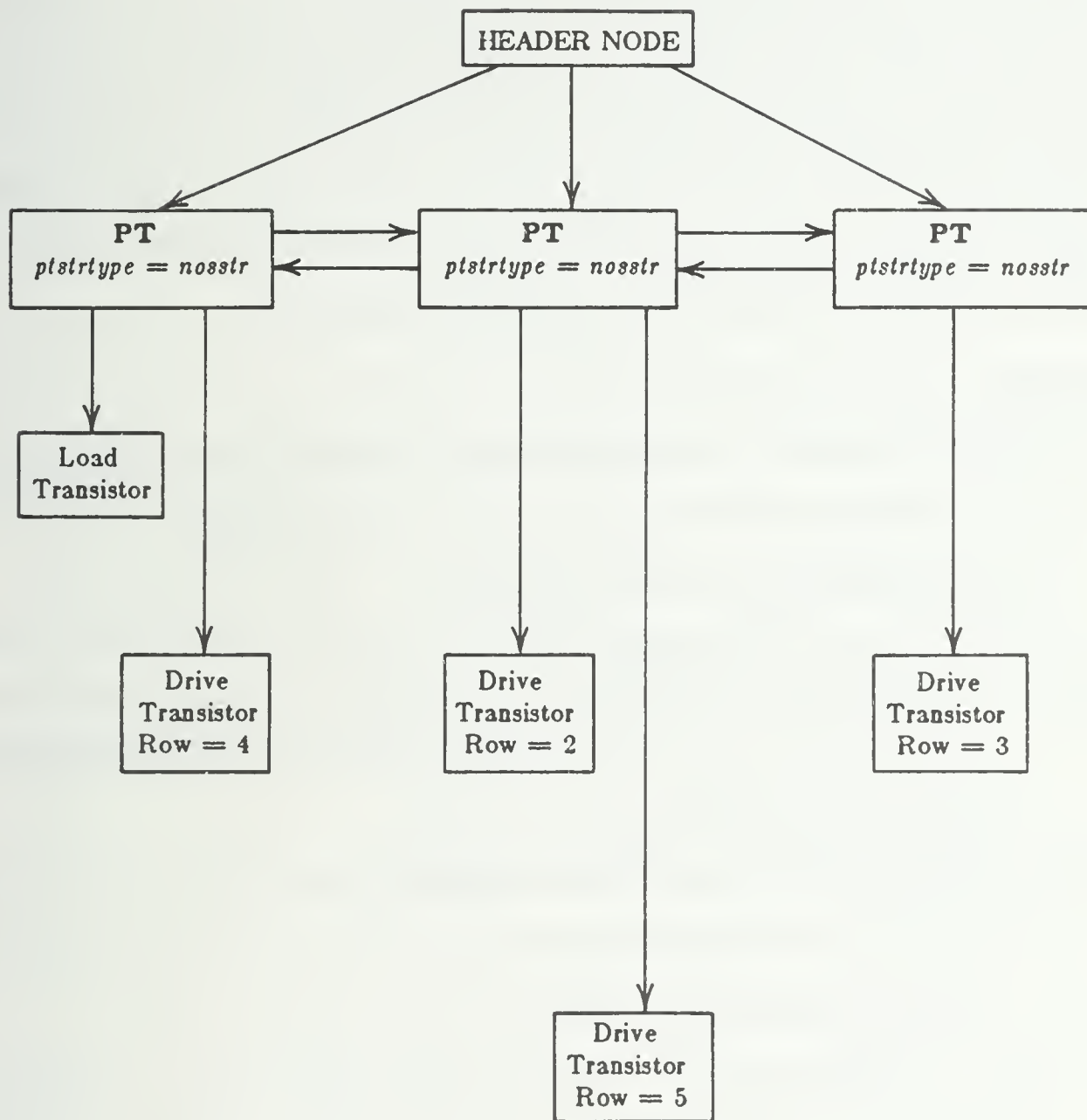


Figure 3.2 Product Term Linked List

3.3. Graphic Representation

The symbolic and geometric representations of a special structure also require special consideration. The routable area layout is taken care of by GEOLAY and SYMBOLAY in the same manner as for normal product terms. However, instead of calling a load transistor macro, the appropriate special structure section macro is called. The x-coordinate of each section macro (and product term) is dependent upon the width of the section immediately preceding it (see section 4.2.2). The y-coordinate of all product terms and section macros of one structure are the same. The symbolic representation of a section is simply an empty outline. All global routing into a section is included in the symbolic layout.

3.4. Parameters

The parameters of the reserved area deal with the location of each section and the transistor sizes. The following is a list of general parameters used by each section of a special structure.

- Respoint* : The upper-left corner of the reserved area section.
- Deltaz* : An integer used to vary the pitch between product terms.
- Drtranswidth* : The width of the drive transistors in the reserved area. Also determines the width of the product term diffusion bar.
- Drtranslength* : The length of the drive transistors in the reserved area.
- Ldtranswidth* : The width of the load transistor channel.
- Ldtranslength* : The length of the load transistor channel.
- Orientation* : The orientation of the section. May presently only be set to zero.

A more detailed explanation of these parameters will be given in section 4.2. As the library of special structures increases, additional parameters may be required.

CHAPTER 4

SPECIAL STRUCTURE DESIGN

The first step in creating a special structure is to design both the routable and reserved areas. All control (select) lines should be placed in the routable area if possible. These lines may then be manipulated by ICE or routed by TCELL. Once a design has been laid out on paper, the structure must be divided into sections following one basic rule: One, and only one, product term must accompany each section. This rule also establishes that, at most, one load transistor and one metal-B bar is allowed per section. An example of the sectioning of a multiplexer is shown in figure 4.1. This structure was created using ICE and will be used as an example of the development of a special structure. In this example, the structure was divided in such a way as to use at most one metal-A to metal-B contact per section for use as a section I/O.

Once the structure is sectioned, a Pascal procedure (macro) must be written for each section. As previously mentioned, these procedures are developed by making calls to the device routines in *ppfall2.h* or to specially designed procedures. The code used for the multiplexer is listed in Appendix A. The listings of the device routine code can be found in the directory

/mntb/1/kubitz/ppf/src/cad

on CRL VAXB. The most frequently used device routine is *maskbox*. This procedure simply draws a rectangle with a specified height and width. Maskbox is called using:

maskbox (<refpoint>, <height>, <width>, <masklevel>, orientation).

The reference point is the upper-left corner of the rectangle. The height and width are defined by using one of the values declared in the design rules (i.e. *wmtla*, *wpoly*, *polypoly*, ...). Masklevel defines the mask level (i.e., *diffusion*, *polysilicon*, *metal-A*, *metal-B*, *implant*, ...) of the rectangle. The mask level also implies the color of the rectangle for display purposes.

One specially developed procedure frequently used in special structures is *geoldtrans.i* (/mntb/1/kubitz/schwieso/lfiles). This procedure is essentially a diffusion bar (product term) and a load transistor. The diffusion bar, unlike a normal product term, does not include a diffusion to metal-A ground (GND) contact. This diffusion bar is used for drive transistors and interconnect within the

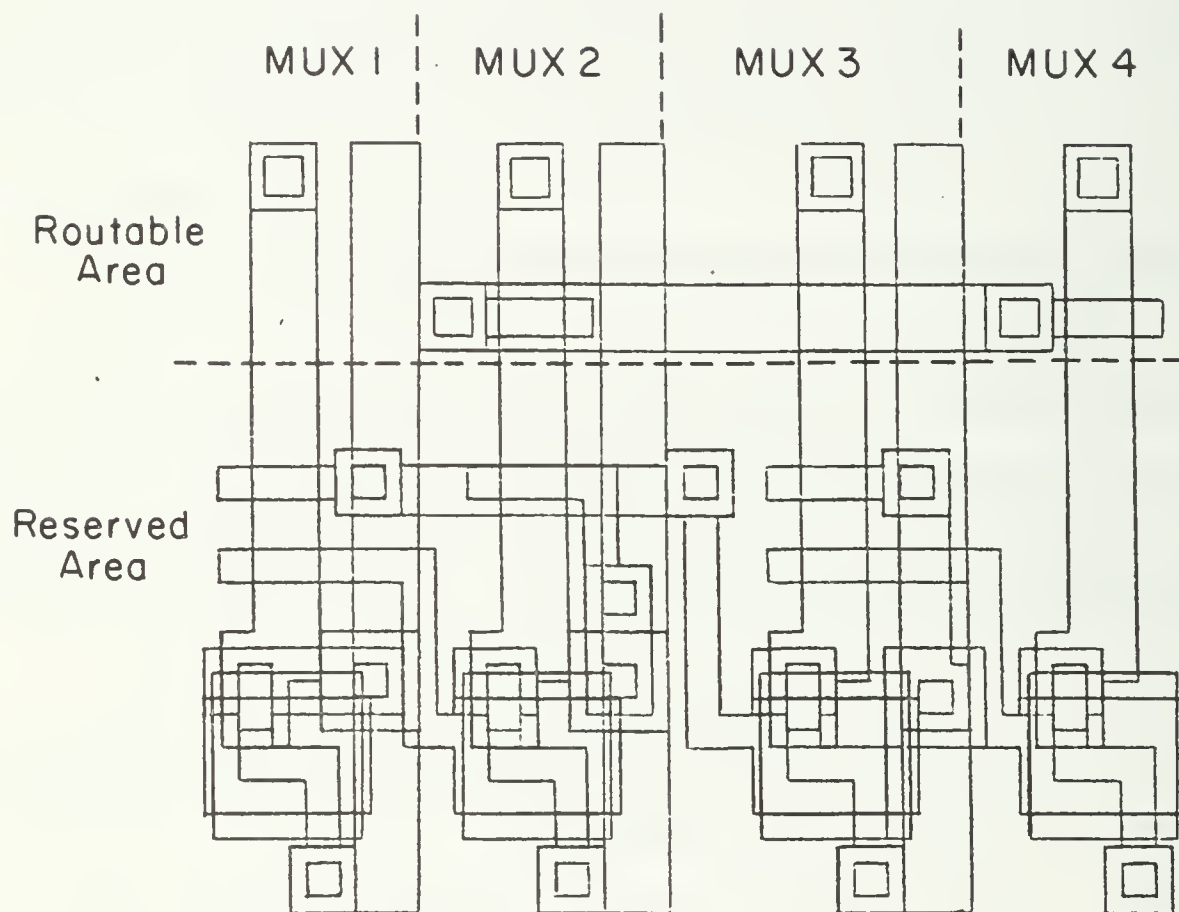


Figure 4.1 2-1 Multiplexer

reserved area. The height of this bar (*diffheight*) is used to determine the difference between the geometric height of a load transistor and the height of the special structure. This height is also stored as the special structure height (i.e. *muxheight*) in *ddefcons.i*. This special structure height must be an integer multiple of the minimum pitch of the cell grid. The minimum grid pitch is $8\text{-}\lambda_{\text{min}} ((2 * \text{pmcutclear}) + \text{wfirstcut} + \text{mtlbmtlb})$. The height and width of the diffusion bar, along with the normal load transistor parameters, make up the parameters to *geoldtrans.i*. This macro, therefore, may be used instead of the normal load transistor macro without making any changes to the load transistor node in the data structure.

4.1. Section I/O Placement

Section I/O contacts are located within the reserved area of a special structure section. A section I/O node is used to allow a connection for metal-B routing (or interconnect) within the reserved area of a special structure. As previously mentioned, each section of a special structure contains only one product term. Each product term is associated with one cell I/O node, which in turn, is associated with one metal-B column. Consequently, only one I/O contact per section is permitted. Only one section I/O contact should ever be required since only one load transistor may be present per section and all input connections (metal-B to drive transistor) may be placed in the routable area of a section. Whenever an I/O contact is not required for the load transistor output, it may be used to connect a metal-B bar to a drive transistor. In this way the section I/O contact may be used to connect an input variable to the structure within the reserved area. The input variable placement must be determined by the designer during the initial layout phase of the structure design (recalling the design objective of keeping the height of the reserved area as small as possible). For example, the drive transistor to metal-B contact for one of the input variables of the mux special structure was inserted into the reserved area of the second section (see figure 4.1). Although a slight increase in the width of the second section was required to allow the insertion of this section I/O contact, if this input were to be inserted in the routable area an additional row would be required which would increase the height of the entire cell. Thus, inserting a section I/O contact for the metal-B to drive transistor connection within the reserved area actually reduced the overall area required by the special structure.

A section I/O contact is represented by a metal-A to metal-B contact within the reserved area. The y-coordinate of this contact must be near the y-coordinate of a row in the cell grid. This is required because a metal-B bar may only end on an I/O node or a metal-A node. Each metal-A node must lie along one of the rows of the cell metal-A grid. Some flexibility, however, is allowed because metal-A may be laid out anywhere in a handcrafted section macro as long as the design rules are not violated. Only two requirements restricted the placement of a section I/O contact: no metal to metal design rules may be violated and the section I/O contact must abut with the metal-B I/O bar. The y-coordinate of the metal-A node (which is used as an endpoint for the metal-B I/O bar) and the y-coordinate of the section I/O contact do not need to be identical as long as the metal-B bar and the section I/O contact abut when the metal-A node is used as an endpoint for the metal-B bar. If the y-coordinates differ significantly, a metal-B bar may be inserted into the section macro. The I/O metal-B bar will then need only abut with this metal-B bar. This will eliminate the possibility of the metal-B I/O bar ending before abutting with the contact or extending beyond the contact. In general, however, all section I/O contacts should be placed as close as possible to the intersections of the cell metal-A and metal-B grids to eliminate the possibility of metal-B bars which extend beyond the boundaries of the section I/O contact and do not abut with any other node. The y-coordinates of the metal-A rows may be determined by locating the butting contact of the load transistor. This butting contact is always the second to last row in the grid (the last row is always the VDD power bar which is located along the bottom edge of the cell). A constant pitch of 19-lambda is used between the VDD bar and the butting contact of the load ($w_{contact} + mtlamtl + w_{firstcut} + dmcutclear + (2 * wdiff) + (2 * polygateclear) + diffpoly$). All other rows may be located by increasing the y-coordinate by the vertical pitch of 8-lambda ($w_{contact} + mtlbmtlb$).

This amount of flexibility, however, is not allowed for the x-coordinate of a contact. The contact not only must abut with the proper metal-B bar but must also not abut with any adjacent metal-B bar. The pitch between the contact and an adjacent metal-B bar must be 4-lambda ($mtlbmtlb$) in accordance to the design rules. The right-most boundary of any metal-B bar always lies along the x-coordinate of an I/O node. The right-most edge of an I/O contact, or any metal-A to metal-B contact, must lie along an I/O node column. The right-most boundary of every special structure section will lie along an I/O node column. This is due to the method by which GEOLAY lays out the columns of a cell. Each section I/O

contact must, therefore, lie within a *wmtlb* distance of the section boundary (see figure 4.2). The location of the contact should always remain fixed in relation to the section boundary. If a section is expanded, due to increases in the channel length of the load transistor, the x-coordinate of the contact should be increased appropriately. All interconnect abutting with the contact must also be expanded or shifted appropriately. If the section is to be expanded to allow global metal-B routing alone (no expansion due to increases in the channel length) the position of the I/O node will not be shifted. Consequently, the position of the section I/O node must not be shifted. For example, if an I/O node (required for global metal-B routing) is desired between section A and section B, the x-coordinate of the I/O node associated with section A will not be changed. The section I/O contact in section A (and all interconnect abutting to the contact) must not be shifted. However, all interconnect which must abut with section B must be expanded. In general, the contact should always be placed as close to the boundary as possible to allow additional global metal-B routing, as described previously, without violating design rules.

4.2. Parameters

The set of parameters used by the multiplexer was described in section 3.4. A minimum set of parameters would include: *respoint*, *drtranswidth*, *drtranslength*, *ldtranswidth*, *ldtranslength*, and *deltax*. These parameters directly affect the transistor sizes and the location of the sections. The transistor sizes must be variable so that they may be modified to adjust the electrical characteristics to match speed/power requirements.

4.2.1. Device Sizes

The channel width of the load transistor may be set using the *ldtranswidth* parameter. This channel width is the width of the diffusion bar defining the source and drain of the load transistor. This parameter does not directly affect the actual geometric width (using orientation "1" which lays out horizontal VDD and GND bars as in figure 4.1) of the load transistor. The load transistor macro was designed to allow slight variations in the channel width without requiring any variation to the geometric size of the load transistor. A maximum value for the channel width, therefore, has been predefined so that the channel width will never have to vary the geometric size. This maximum value has been set to 3-lambda ($3 * \lambda$)

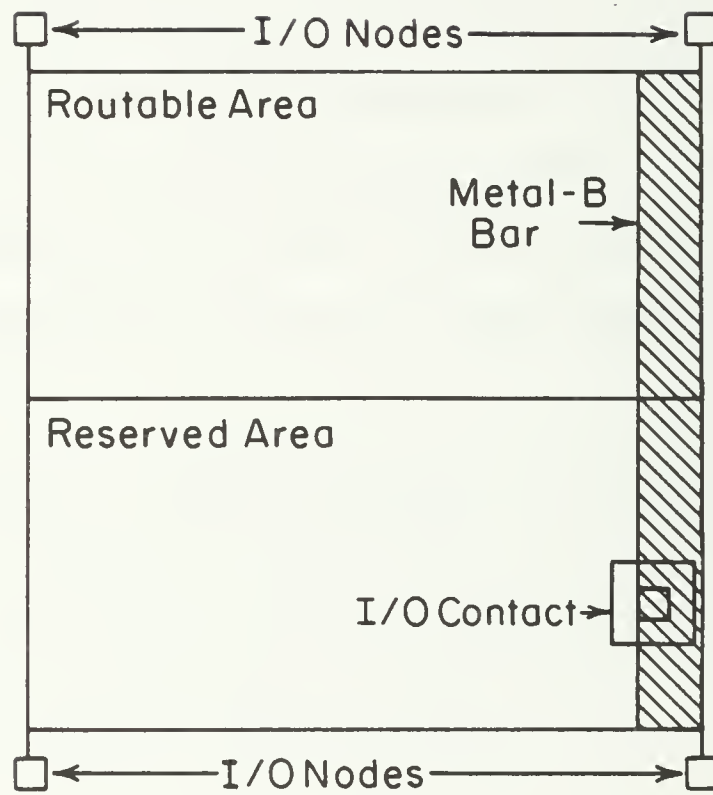


Figure 4.2 I/O Contact Placement In Special Structure Section

wdiff). Unlike the channel width parameter, variations in the channel length will directly affect the geometric size of load transistor. The channel length of the load transistor is set using the *ldtranslength* parameter. As the channel length parameter increases, the geometric width of the load transistor (see figure 4.3) will increase. As previously mentioned, the geometric height of the load transistor is fixed and is not affected by either the channel length or channel width. Possible increases in the geometric width of a load transistor must be taken into account as the section is designed. The width of a special structure section must be expandable to allow for variations in the geometric width of a load transistor. A *deltax* variable was used by each section procedure to allow variations in the total width of the section. In addition to being an internal variable, *deltax* has also been declared a parameter to the section macro. This gives the user the capability of manually setting the section width (see section 4.2.2). Each section procedure will compare the total section width, taking into account any width additions due to the *deltax* parameter, with the section width required to accommodate the geometric width of the load transistor and make any necessary expansions.

A maximum value for the channel length, at which the rightmost boundary of the load transistor matches the rightmost boundary of the section (assuming orientation "1" as in figure 4.1), must be calculated for each section. Any value of channel length beyond this maximum value must be added to *deltax* (internal) to increase the width of the section. However, if a value for *deltax* (parameter) has been passed to the procedure, this maximum value for the channel length must be updated. The updated value would be the sum of the previous maximum channel length value and the value of *deltax* (parameter). If the channel length exceeds this new maximum, the value of *deltax* (internal) must be increased. A reasonable initial maximum value may be estimated but may require updating once the section is laid out and tested. The pitch between the I/O nodes adjacent to the section and the x-coordinate of the section I/O contact must then be increased by the value of *deltax* (internal). All interconnect which must abut with the next adjacent section must be expanded by the maximum value of *deltax* (be it the parameter or the internal value).

Parameters have also been established which allow the user to set the sizes of the drive transistors in the reserved area. The width of the drive transistors, along a given product term, may be varied using *drtranswidth*. This parameter determines the width of the product term diffusion. In TCELL, the width

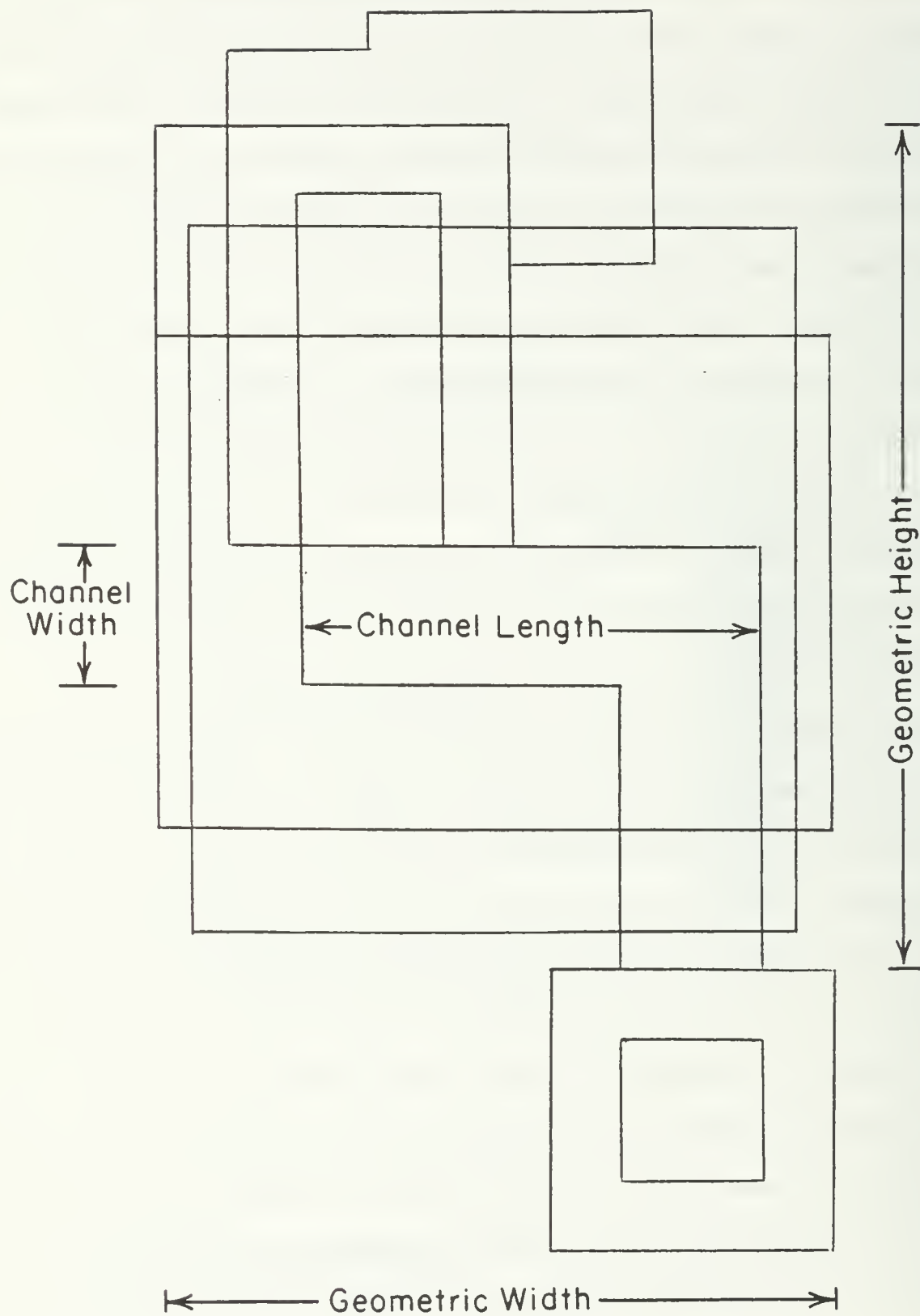


Figure 4.3 Load Transistor Macro

of all drive transistors along a given product term is a constant determined by the number of drive transistors along that product term. TCELL will also vary the width of the drive transistors to satisfy the drive capability requirements. However, if the width of one drive transistor along a given product term is increased, using ICE, the width of all drive transistors along that product term will increase proportionally. The width of the drive transistors in the routable area, therefore, directly affects the width of the drive transistors in the reserved area. The same variable used for the diffusion bar width in the routable area is used as the *drtranswidth* parameter for the special structure sections. The length of the drive transistors within the reserved area is also variable. The length of all drive transistors in the reserved area are set using the parameter *drtranslength*. A maximum value of $4\text{-}\lambda$ ($2 * w_{poly}$) was determined for this length. Beyond this value the length begins to exceed the contact length. Consequently, if two drive transistors are present along one product term in a reserved area, the polysilicon to polysilicon design rule (*polypoly*) may be violated (see the mux1 section in figure 4.1).

4.2.2. Section Placement

The parameters used for placement information are *retpoint* and *deltax*. The *retpoint* is used to position the (reserved area) section macro such that the proper abutment of product terms and sections occurs. For the multiplexer the reference point of each section is the uppermost point along the left edge of the product term diffusion bar (see figure 4.4).

As mentioned previously, *deltax* is used to expand the width of a section. This parameter was established to allow the designer the option of increasing the distance between adjacent product terms within a special structure. Additional cell I/O nodes may then be inserted to allow global metal-B routing through the structure. *Deltax* is also used as a variable within each section. All expansions and shifts of rectangles within a section must be toward the right boundary only. Using right edge expansion makes section placement by GEOLAY and SYMBOLAY much less complicated because of the way these procedures lay out each section. The initial placement of the right edge boundary is determined using initial values of the device sizes. Any shifts or expansions due to increases in devices sizes, which may cause a rectangle to exceed the right boundary, are determined and added to *deltax*. *Deltax* then extends the right boundary and extends all rectangles which must abut with the right boundary accordingly. The sec-

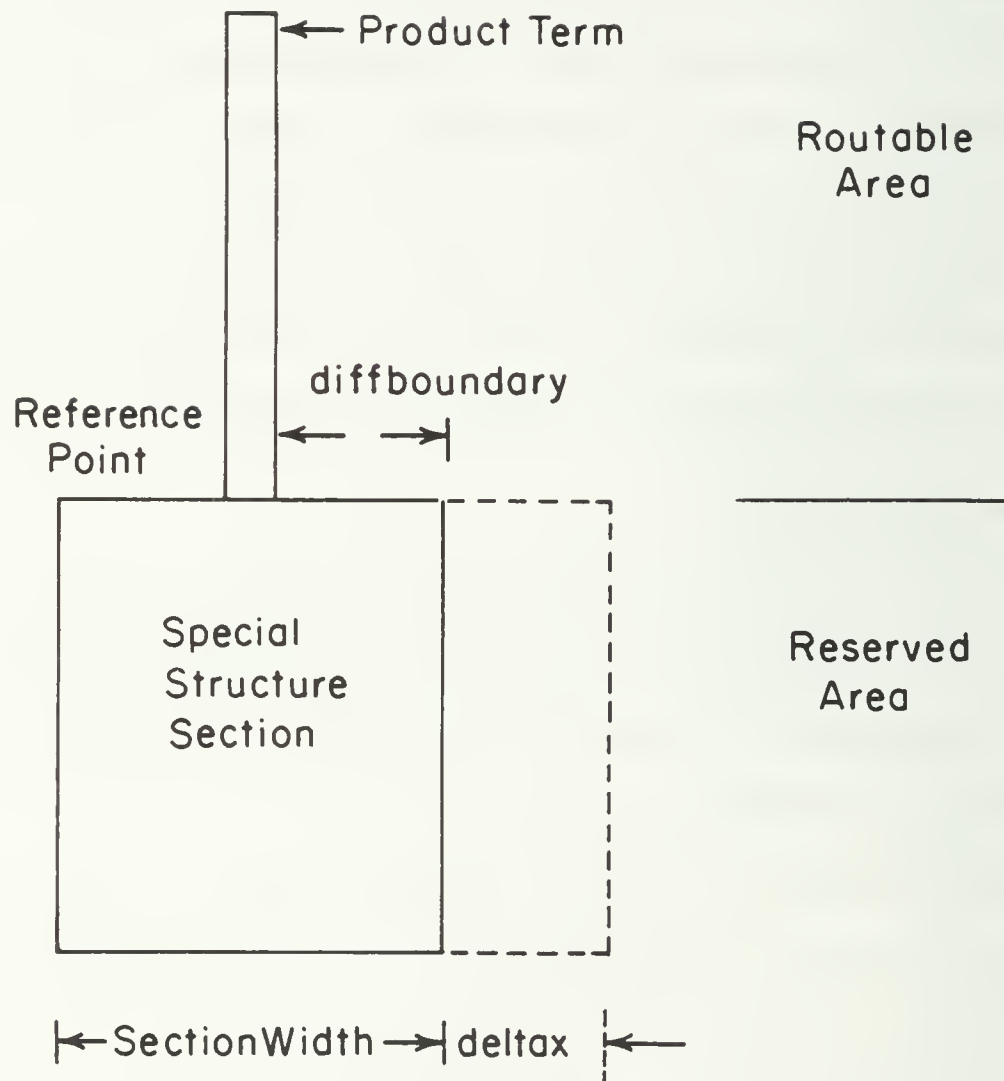


Figure 4.4 Section Macro Reference Point

tion I/O contacts are also shifted to assure proper alignment with the appropriate metal-B bar. As mentioned in section 4.2.1, section I/O contacts are shifted only when the width of the column needs to be expanded due to increases in the channel length of the load transistor. In this way all interconnect between sections, and routing to each section, will retain proper abutment. In addition, this method assures that load transistors will never overlap but remain the minimum distance apart as determined by the design rules. The minimum distance between load transistors is normally established by the polysilicon to polysilicon pitch (*polypoly*). The updated value of Δx is then passed back, from the section procedure, to the layout procedure. The layout procedure may then adjust the column width accordingly. The increase in the column width will assure that VDD, GND and other metal-A bars abut correctly from column to column. The correct placement of the next column will also be assured. Section placement by GEOLAY is discussed in further detail in section 5.3.1.

In general, a great deal of flexibility is allowed in the design of special structures. Although the height of the reserved area sections must be kept at a minimum and must remain constant throughout the structure, the width of each section is variable and may be set to whatever the designer requires. Most any type of interconnect is allowed within the reserved area, with the possible exception of horizontal metal-B, as long as the design rules are upheld. Although horizontal metal-B is possible, in a special structure section, it should be avoided due to the possibility of errors which would occur if a global metal-B bar were to pass through a section and unintentionally abut with a horizontal metal-B bar. The reserved area should allow global metal-B routing but does not necessarily need to allow global metal-A routing. Most reserved areas, in fact, will not allow metal-A routing due to the use of vertical metal-A interconnect. Each section should be expandable to allow the addition of cell I/O nodes which may be required for the insertion of additional global metal-B routing after the special structure is designed.

CHAPTER 5

SOFTWARE MODIFICATIONS

Incorporating special structures into the system required modifications to both the topological data structure and the layout procedures (GEOLAY and SYMBOLAY). Modifications to the data structure require changes to ICE, TCELL and the data definition files. ICE requires additional procedures which may be used to create, delete, or modify a special structure. TCELL requires procedures used to create, place and rout a special structure along with updates to the scan-in procedure which will expand the CDL syntax to allow special structures. This chapter describes the modifications made to the software. Examples from the 2-1 multiplexer, which was incorporated into the system, will be presented to help clarify these software changes and additions.

5.1. Data Structure Modifications

The data structure is defined in the global "type" declarations. All "type" declarations (including "node" declarations) are made in *ddeftype.i*. Consequently, *ddeftype.i* must be updated each time an addition or modification is made to the data structure. This includes all updates needed to add a structure to the special structure library.

5.1.1. Structure Tags

Inserting special structures into the data structure required a modification to the product term node (*prodtermnode*) along with the addition of a "node" for each section of the structure. The addition of a tag declaring the structure type (i.e., random logic, multiplexer, ...) was one of the modifications made to the product term node. This tag is represented as the variable *ptstrtype*. The *ptstrtype* of a product term must be chosen from a list of structure types (*structuretype*) declared in *ddeftype.i*. Each new structure added to the library of special structures, therefore, requires the declaration of a unique structure type. A new structure type may be declared by adding an acronym for the structure to the *structuretype* declaration within *ddeftype.i*. Each section of a given structure may then be identified using the *ptstrtype* tag. For example, the sections which make up the 2-1 multiplexer are each declared type "mux." The structure type tag of a product term is declared in ICE or TCELL at the time the product term is created

using the command:

```
<product term>^.ptstrtype := <structuretype>.
```

5.1.2. Special Structure Nodes

The only other modification to *prodtermnode* was the addition of a pointer (*ptss*) to a special structure section node. If the product term is associated with a special structure, *ptpointer^.ptss* will be used to point to the appropriate section node. Each section within each special structure must be declared as a separate node. In the case of the 2-1 multiplexer the sections are represented as: *mux1node*, *mux2node*, *mux3node*, and *mux4node*. All information pertaining to the reserved area of a section may then be accessed from the appropriate product term node. In the case of a product term associated with a random logic function the *ptss* pointer is set to nil. All routable area information is accessed through the product term node (as is the case with product terms associated with random logic functions).

A listing of the pointers contained in each section node of the multiplexer is shown in Appendix B. Information concerning the reserved area includes integer values for the length and width of the drive transistors and a pointer to a load transistor node. The load transistor node used by the special structures is the same load transistor node used by random logic gates (*loadtransnode*). All information pertaining to the width and length of the load transistor is contained within *loadtransnode*. The pointers used to locate the channel length of the load transistor within the first section (left-most section) of the multiplexer are:

```
ptpointer^.ptss^.mux1load^.llength.
```

A diagram of the nodes and pointers used to access the load transistor node is shown in figure 5.1.

Each section node may also contain a pointer to a metal-B node which is used as an I/O bar for the structure. The I/O bar pointer (i.e., *muzoutmtlb*, *muzin0mtlb*, ...) will point to a metal-B node (*mtlbnode*). This *mtlbnode* contains the information pertaining to the size and location of the I/O bar. The *mtlbnode* is then inserted into the metal-B linked list.

5.2. ICE Modifications

All of the procedures required by ICE are contained in the file "icedemo2.p" located in /mntb/1/kubitz/schwieso/ice on the CRL VAXB. These procedures have also been categorized by their

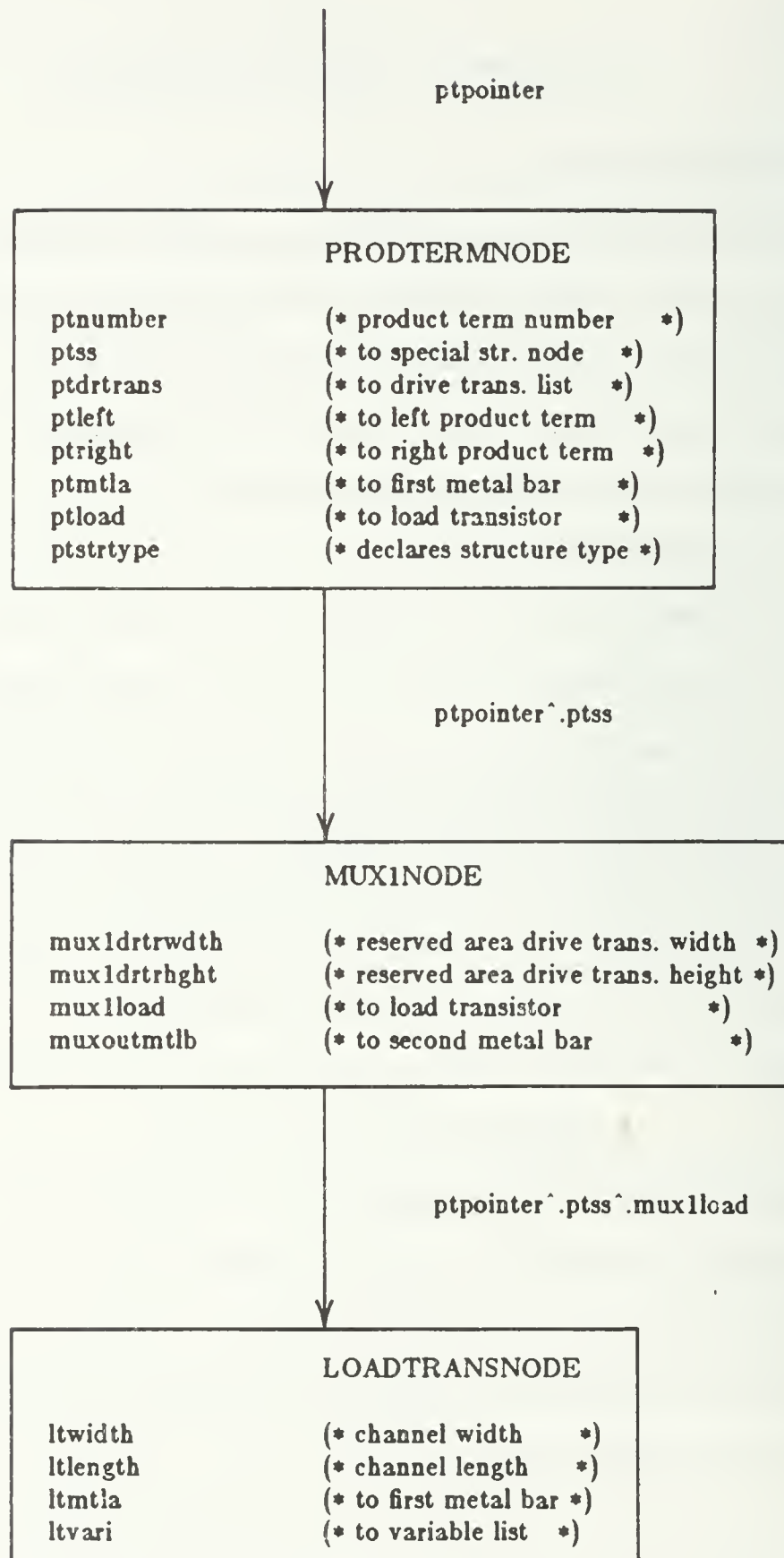


Figure 5.1 Pointers To Load Transistor

functions and split into separate files. For example, all procedures used to create new nodes for a linked list have been placed into a file "create.p." These ICE files are located in directory /mntb/1/kubitz/work/midas/src/ice also on CRL VAXB. These files, rather than icedemo2.p, are used to run ICE in the MIDAS environment. Consequently, all additions to icedemo2.p (and to the data definition files) must eventually be added to the appropriate files containing the ICE source code used by MIDAS.

5.2.1. Menu Entries

Each special structure must have a menu entry in the insert and delete operation menus. A general modify procedure (*modifyss*) has been developed which may be used for any special structure. The modify menu, therefore, does not require updates for each new structure. When a special structure is created, each product term associated with each of the sections is automatically created. Similarly, a delete operation will delete each of the product terms associated with a structure. The modify operation, however, only affects the reserved area of individual sections of a special structure. Modify allows the user to vary transistor sizes within a reserved area along with the capability of changing the endpoints of the metal-B bars used for the section I/O. The *initmenu* procedure is used to display the possible operation menus. The appropriate operation menu may then be chosen (i.e., create, delete, display, ...) and the desired procedure may be initialized and called from the operation menu (i.e., create product term, delete metal-a, ...). Additional entries may be inserted into a menu by copying an existing menu entry into an unused "menulist." An example of the menulist used for creating the mux structure is shown in figure 5.2. The *menu* size must then be increased to account for any additional menu entries. Once the menu entry has been inserted the appropriate operation procedure (i.e., createmode, modifymode, deletemode,...) must be updated to include the addition menu choice. The operation procedure then calls the appropriate procedure(s) required to perform the operation. In the case of MIDAS, however, all menu entry declarations are initialized and called from the midas file *ops.c*. All of the MIDAS procedures referenced are described in greater detail in [Esak83]. This file is located in directory /mntb/1/kubitz/work/midas/src/midas.

```

with menulist[12] do begin
    menuname[0] := 'ENTER' ;
    mcolor[0] := white ;
    msize[0] := 5 ;
    menuname[1] := 'MUX' ;
    mcolor[1] := white ;
    msize[1] := 3 ;
    menuname[2] := ' ' ;
    mcolor[2] := black ;
    msize[2] := 0 ;
    shortname := 'ENTER MUX' ;
    msize[3] := 9 ;
end ;

```

Figure 5.2 Menulist

In the case of the insert special structure operation, MIDAS should list the entire library of special structures available. This may be done by either creating a new menu or listing the library on the terminal screen. The user may then indicate the desired special structure so the appropriate procedure may be called. For example, if a 2-1 multiplexer needs to be inserted the procedure *insertmux* must be called. Presently, all modifications have been made only to *icedemo2.p* and the source code used by *icedemo2.p*.

5.2.2. Procedures

Once the menu entries have been implemented, separate insert and delete procedures must be written for each special structure. The modify procedure must also be updated to include any new structure. The insert procedure is normally the most complex of the operation procedures required for a special structure because each of the product terms, along with each of the transistors and metal nodes, must be created, initiated and inserted into the linked lists. The delete operation simply deletes each of the nodes, associated with the structure, from their appropriate linked list. The modify procedure must locate the appropriate section node associated with the desired product term. Once located, any of the device sizes in the reserved area may be modified by updating the appropriate length, or width pointers. Device sizes

in the routable area may be modified by simply choosing the desired operation in the modify menu (i.e., modify drive transistor, modify metal-A, ...). The modify procedure may also be used to change the end-points of the I/O metal-B bar associated with a section.

An operation procedure for one special structure may be somewhat different than the same operation procedure used by a second special structure due to variations in the number of sections required and the number of transistors in the routable area of each section. The use of a general insert and delete procedure, used by any special structure, would be much more useful than would a separate set of procedures for each structure. However, a general procedure would be much more complex to program and to update (to include future structures) and would not be any more efficient than would separate procedures. As an example of an insert operation, the procedures used to insert the 2-1 multiplexer into the data structure will be described. These descriptions may give a designer insight into producing algorithms for inserting additional special structures.

Additional procedures were required to implement the 2-1 multiplexer insert procedure. These procedures include: *insertmux*, *newssload*, *makedtnode*, *insertssionode*, and *setssiomb*. Most of these procedures are simply modifications to existing procedures. Each of the procedures, excluding *insertmux*, were designed such that they may be used for any special structure. The procedure *newssload*, however, requires slight modifications (additions to a case statement) for each new special structure node. *Insertmux* is the primary procedure used to create the multiplexer. The only parameter passed to *insertmux* is the product term number of the first mux section. The product term number is used to determine the location of the entire mux structure within the cell. The following is a description of the *insertmux* procedure.

```

procedure INSERTMUX:
begin
  While number of rows is less than minimum number of rows
  required for mux do
    Insert new I/O node.
  Set loop counter to 1.
  While loop counter not greater than no. of mux columns do
  begin
    Create product term node (newptptr).
    Insert product term node into product term list.
    Set structure type of product term to mux.
    Create metal-A node (newmaptr).
  end
end

```

Create special structure node (ssptr).

Case loop counter of

1: begin

Set special structure node tag to mux1node.

(* Reserved Area Data *)

Perform newssload.

Enter length of drive transistors.

Set length pointer in special str. node (mux1drtrlgth).

Enter width of drive transistors.

Set width pointer in special str. node (mux1drtrwidth).

(* Routable Area Data *)

Perform insertssionode (muxoutmtlb).

Set output metal-B pointer in special structure node.

Have special structure pointer in product term node (ptss)
point to special structure node.

end.

2: begin

Set special structure node tag to mux2node.

(* Reserved Area Data *)

Perform newssload.

Enter length of drive transistors.

Set length pointer in special str. node (mux2drtrlgth).

Enter width of drive transistors.

Set width pointer in special str. node (mux2drtrwidth).

(* Routable Area Data *)

Perform insertssionode (muxin0mtlb).

Set input metal-B pointer in special structure node.

Perform makedtnode (muxdrtr).

Set width pointer in special str. node (mux2drtrwidth)
to width of drive transistor created in makedtnode.

Have special str. pointer in product term node (ptss)
point to special structure node.

end.

3: begin

Set special structure node tag to mux3node.

(* Reserved Area Data *)

Perform newssload.

Enter length of drive transistors.

Set length pointer in special str. node (mux3drtrlgth).

Enter width of drive transistors.

Set width pointer in special str. node (mux3drtrwidth).

(* Routable Area Data *)

Perform insertssionode (muxin1mtlb).

Set input metal-B pointer in special structure node.

Have special str. pointer in product term node (ptss)
point to special structure node.

end.

4: begin

Set special structure node tag to mux4node.

(* Reserved Area Data *)

Perform newssload.

(* Routable Area Data *)

Perform makedtnode (muxdrtr).

Create metal-A node for select bar (muxmaptr).

Enter width of metal-A bar.

```

Set left column of metal-A bar to product term
containing first routable area drive transistor in mux.
Set right column to previous product term.
Set row number of metal-A bar to row immediately above
reserved area (same row number as drive transistors).
Perform insertmanode.
Have special str. pointer in product term node (ptss)
point to special structure node.
end.
end.
Increment loop counter.
Increment product term number.
end. (* while *)
end.

```

Procedure `newssload` is used to create a load transistor node for a special structure section. This procedure sets the channel width and channel length of the load transistor. A case statement is then used so that the appropriate load transistor pointer associated with the section node points to the newly created load transistor node. The parameters passed to `newssload` are the special structure pointer and the tag identifying the special structure section.

```

procedure NEWSSLOAD:
begin
Create load transistor node (ltptr).
Case special structure section of
mux1node: begin
Have special structure node (ssptr^.mux1load)
point to load transistor node.
end.
mux2node: begin
Have special structure node (ssptr^.mux2load)
point to load transistor node.
end.
(* Repeat node declarations for each sp. str. section *)
end. (* case *)
Enter channel width of load transistor.
Enter channel length of load transistor.
end.

```

Procedure `makedtnode` is simply a copy of the set of commands used by the `createmode` procedure to create a drive transistor. A special procedure had to be developed because the drive transistor create operation procedure can only be accessed from the `createmode` menu entry. The parameters to `makedtnode` are the pointer to the product term, the row number for the drive transistor, and the product term number.

```

procedure MAKEDTNODE:
begin
  Create drive transistor node (dtptr).
  Find product term associated with product term no. (ptptr).
  If product term does not exist
  then output 'no such product term'
  else begin
    Enter drive transistor width.
    Enter drive transistor length.
    Perform insertdt. (* Inserts drive transistor node
    into drive transistor linked list and sets
    appropriate pointers *)
  end.
end.

```

Makedtnode is used to create drive transistors for the routable area of a special structure. Consequently, these nodes are pointed to from the product term node alone (not the special structure section node) and are inserted into the drive transistor linked list. No special procedures are required to edit these routable area transistors.

All input and output metal-B bars are inserted using insertssionode. The parameters to insertssionode are the product term pointer, the row number of the section I/O contact within the reserved area, a metal-B pointer, and a flag (setwidth) used to indicate whether the width of the metal-B bar needs to be set. When insertssionode is called from an insert structure procedure this width must always be true. If insertssionode is called from a modify procedure the width may not require setting. Metal-A and metal-B nodes are created by insertssionode. This is required because each end point of a metal-B bar must point to either an I/O node or a metal-A node. A metal-A node, therefore, must be created to represent the section I/O contact within the reserved area. Although this metal-A node is inserted into the metal-A linked list, the metal-A bar will not be laid out. The metal-A bar need not to be laid out because all metal-A nodes in the reserved area of a special structure are laid out as part of each section macro. The layout of metal-A nodes within a reserved area is avoided due to a modification made to GEOLAY (section 5.3). The user also sets the endpoints of the section metal-B I/O bar from insertssionode. These endpoints may either be the top and bottom I/O nodes, the top or bottom I/O node and the section I/O contact, the bottom I/O node and any metal-A bar in the routable area, or the section I/O contact and any metal-A bar in the routable area. If the metal-A bar is already present, the appropriate pointers to the metal-A node are set. Otherwise, a new metal-A node is created and the user

must indicate the endpoints. The metal-B node is then inserted into the metal-B linked list.

procedure INSERTSSIONODE:

```

begin
  Create metal-A node (maptr).
  Create metal-B node (mbptr).
  Enter width of metal-B bar.
  Enter top and bottom end points of metal-B (I/O) bar.
  If metal-B bar extends to top and bottom I/O nodes
  then iocase = 1.
  If metal-B bar extends to top I/O node and I/O contact
  in reserved area
  then iocase = 2.
  If metal-B bar extends to bottom I/O node and I/O contact
  in reserved area
  then iocase = 3.
  If metal-B bar extends to I/O contact in reserved area
  and a metal-A node in routable area
  then iocase = 4.
  Find I/O node along top edge (topptr).
  Find I/O node along bottom edge (botptr).
  Case iocase of
  1: begin
    Set metal-B "top" I/O pointer to top I/O node.
    Set metal-B "bottom" I/O pointer to bottom I/O node.
  end.
  2: begin
    Set metal-B "top" I/O pointer to top I/O node.
    Set metal-B "bottom" metal-B to metal-A pointer to
    metal-A node.
    Set metal-A "up" metal-A to metal-B pointer to
    metal-B node.
  end.
  3: begin
    Ask if I/O bar terminates at a metal-A node other
    than at the I/O contact.
    If yes
    then begin
      Enter row number of metal-A node in routable area.
      Find metal-A node.
      If no metal-A node found
      then perform newmanode.
      Set metal-B "top" metal-B to metal-A pointer to
      metal-A node.
    end
    else set metal-B "top" metal-B to metal-A pointer to
    metal-A node.
    Set metal-B "bottom" I/O pointer to bottom I/O node.
    Set metal-A "down" metal-A to metal-B pointer to
    metal-B node.
  end.
  4: begin
    Set metal-B "bottom" metal-B to metal-A pointer to
    metal-A node.

```

```

Set metal-A "up" metal-A to metal-B pointer to
metal-B node.
Ask if new I/O node is required.
If yes
then perform newionode.
Enter row number of metal-A bar in routable area.
If row number entered is in reserved area
then ask for another row number.
Find metal-A node (tptr).
If no metal-A node found
then perform newmanode.
Set metal-B "top" metal-B to metal-A pointer to
routable area metal-A node.
Set routable area metal-A node "down" metal-A to
metal-B pointer to metal-B node.
end.
end. (* case *)
Perform insertmanode.
Set column number to product term number.
Insert the metal-B node into the metal-B linked list.
(* The rest of insertssionode is the same as the procedure *)
(* insertmbnode (inserts metal-B node into linked list). *)
(* The only modification was that instead of calling *)
(* setiombptrs, the procedure setssiomb is called. *)
end.

```

The procedure *setssiomb* is essentially the same as the procedure *setiombptrs*. The only difference is that an additional parameter was added to *setssiomb*. This parameter ("iocase") is used to determine if the metal-B bar extends to I/O node or not. If *setiombptrs* were used, the user would be asked if the metal-B bar extended to the I/O node. This question, however, is unnecessary (and redundant) because this information was already input in *insertssionode*.

Except for the addition of the procedures described, the original ICE procedures have not been significantly modified. The only other change was the addition of a parameter to the *maerrchk* procedure. This procedure is used to determine if an error has occurred due to unintentional overlapping of metal-A bars. When the metal-A nodes required for the I/O contacts were inserted into the metal-A linked list (procedure *insertssionode*) an overlap error was flagged. The error, however, may be ignored because the I/O contact metal-A nodes are not actually laid out in GEOLAY. A flag (*errorchk*) was added to the *maerrchk* parameters which may be used so that the overlap flag will not be set. If this *errorchk* flag is set, the metal-A nodes will be checked for overlaps as usual. Additions to ICE will always have to be made as long as the library of special structures increases. No major modifications to ICE, however, should be required to support future special structures.

5.3. Geolay

A number of modifications were also required to the procedures which make up GEOLAY. GEOLAY is made up of five basic procedures. *Geolay* is the main procedure and is used to obtain the information required to lay out each column and to initialize the variables, constants, and coordinates. *Columnlay* is called from *geolay* to actually lay out each column. *Drtranslay*, *muzlay*, and *mtlblay* are in turn called by *columnlay* to lay out the drive transistors, mux special structure, and metal-B bars respectively.

```

procedure GEOLAY:
begin
  Enter initial point.
  Enter model number.
  Initialize start points.
  Initialize constants.
  Set section counter to 1.
  Initialize y-coordinate of each row.
  Set product term pointer to first product term node.
  Set column pointer to first I/O node.
  While product term pointer not nil and
  column pointer not nil do
  begin
    Set column counter to I/O number.
    Set structure type (strtype).
    Set compensation height (compheight).
    If product term number = column counter
    then set product term exist flag (ptexist).
    Perform columnlay.
    If product term exists and product term pointer not nil
    then set product term pointer to next product term.
    If column pointer not nil
    then set column pointer to next column.
  end.
  Output size of cell.
end.

```

Columnlay is the procedure which lays out each column of the cell. Significant changes were required to *columnlay* so that the special structure sections could be laid out. Each new special structure will require small additions to *columnlay*. The following is a description of *columnlay*.

```

procedure COLUMNLAY:
begin
  If a product term exists
  then begin
    Set drive transistor pointer.
    Set product term diffusion bar width.
  end
end.

```

Case structure type of

nosstr: begin

Set pitch between left and right I/O nodes adjacent to column.

Set pitch between product term and right I/O node (diffboundary).

end.

mux: begin

case section counter of

1: begin

If diffusion bar width too small for width of drive transistors in reserved area then increase diffusion bar width.

Set pitch between left and right I/O nodes adjacent to column (to section width).

Set diffboundary pitch.

end.

2: begin

If diffusion bar width too small for width of drive transistors in reserved area then increase diffusion bar width.

Set pitch between left and right I/O nodes adjacent to column (to section width).

Set diffboundary pitch.

end.

3: begin

If diffusion bar width too small for width of drive transistors in reserved area then increase diffusion bar width.

Set pitch between left and right I/O nodes adjacent to column (to section width).

Set diffboundary pitch.

end.

4: begin

Set pitch between left and right I/O nodes of column to section width.

Set diffboundary pitch.

Set flag declaring last product term in special structure.

end.

end.

end.

else set pitch between left and right I/O nodes.

Set x-coordinate of start point and next start point.

Set row pointer to top left header node.

Layout GND metal-A bar.

Initialize metal-B bar.

If product term exists

then begin

If diffboundary > 0

then adjust x-coordinate of product term diffusion bar.

Layout GND to product term contact.

end.

For each row in routable area do

begin

```

Advance row pointer.
If product term exists
then if not last row in routable area
    then perform drtranslay
else if structure type is special structure
    then begin
        Calculate deltax.
        Case structure type of
        mux: begin
            If section counter not greater than
            number of mux columns
            then perform muxlay.
        end.
        (* Repeat for each special structure *)
        end.
        Increment section counter.
        Adjust x-coordinate of next start point
        due to deltax.
        Increase pitch between left and right
        I/O nodes by deltax.
        end.
    else if no load transistor required
        then layout diffusion track between product terms
    else layout load transistor.
end.
Set row pointer to top left header.
Set x-coordinate for metal-B bar.
For all rows between VDD and GND do
begin
    Advance row pointer.
    If needmtla
    then begin
        If row is in routable area
        then layout metal-A bar.
    end.
    If row is in routable area
    then set flag so no metal-A to metal-B contacts are laid out
    else set flag so contacts are laid out.
    If needmtlb
    then perform mtlblay.
end.
Layout metal-A bar for VDD.
Layout last metal-B bar if required.
Layout additional length of metal-A GND bar
created by addition pitch due to deltax.
end.

```

The variable grid system allows a variable pitch between I/O nodes along either the left-right and/or top-bottom boundaries of a cell. The horizontal pitch (between adjacent I/O nodes along the top-bottom edge) must be varied to allow the insertion of special structure sections. This pitch (*wterm*) is dependent upon the width of each individual section of the special structure. Separate values for *wterm*

must also be established for random logic product terms and the case when no product term exists. If a product term exists, the structure type and section counter are used to determine the section to be laid out. The horizontal pitch between the two I/O nodes along the left and right boundary of the section is then set to the width of that section. A constant, for each section, is also required to correctly abut the product term diffusion bar in the routable area and the diffusion bar in the reserved area. This constant value is represented as `<section>db` (i.e. `mux1db`) (see figure 4.4). This value, along with the initial value for the section width, is declared as a constant in `ddefcons.i`. *Diffboundary* is the variable used to adjust the x-coordinate of the product term. *Diffboundary* is the distance from the the x-coordinate of the right-most I/O node (section boundary) to the product term. This *diffboundary* value is set to the appropriate `<section>db` constant.

Once the appropriate section has been determined, `columnlay` sets the appropriate `wterm` and *diffboundary* values. Once these values are determined, the x-coordinates for the product term and the starting point of the next I/O node may be established. The x-coordinate of a metal-B bar, in reference to its associated I/O node, does not vary from one structure type to another. The right edge of a metal-B bar always lies along the x-coordinate of an I/O node.

Once all of the coordinates have been established, `columnlay` begins to lay out the product term. Each row is laid out until the row number lies in the reserved area of a structure or the load transistor row is reached. All diffusion bars and drive transistors within the reserved areas are laid out by the special structure macros. Each row is laid out by making calls to the procedure `drtranslay`. This procedure determines if a drive transistor is required. If required, `drtranslay` will locate the appropriate drive transistor node and lay out the drive transistor and the product term diffusion bar. If no drive transistor is required, the diffusion bar alone is laid out. If the product term is part of a special structure, `drtranslay` will add a section of metal-A from the drive transistor contact to the left-most boundary of the section. This is required because if the drive transistor is also the right endpoint for the metal-A bar, the metal-A bar will end at the right-most boundary of the previous section. Consequently, if the polysilicon to metal-A contact of the drive transistor does not lie adjacent to the left-most boundary of the section, a gap will appear between the metal-A bar and the contact. All drive transistors associated with random logic product terms abut to the left-most boundary of the section. However, this may not be the case

with special structure product terms. A metal-A stub between the boundary and the contact will then be required for proper abutment to occur.

The height of diffusion bar section laid out per row is the vertical pitch between the two adjacent left-right I/O nodes. Each row is laid out beginning from row number two (row one is always used for the GND metal-A bar and is laid out separately) until the row number equals (totalrows - 1 - compensation height). In the case of random logic product terms, the compensation height (compheight) is zero. Therefore, at row number (totalrows - 1) either the load transistor or a diffusion track connecting product terms is laid out. The last row of each cell (row number equals totalrows) is reserved for the VDD metal-A bar.

In the case of special structures the compensation height is the difference between the geometric height of the random logic load transistor and the height of the reserved area of the special structure. This height is simply the height of the product term diffusion bar used in the special structure macro. The load transistor used in special structures is the same height as the random logic load transistor. Consequently, the vertical pitch between (headerbase^.totalrows - 1) and the last row may remain constant. No modifications, therefore, were required to the row initialization process. *Compheight* is an integer value of the vertical pitch of the cell (i.e., $wcontact + vertpitch$). The compensation height is constant for each section of a particular special structure. The actual height of the special structure section, therefore, is the sum of the vertical pitch required for the load transistor and the compensation height. Each special structure must have a compensation height value stored in *ddefcons.i* (i.e., *muxheight* is used as the multiplexer constant) which is used to determine the layout coordinates of the section macros.

When the first row in the reserved area is reached, the special section macro must be laid out. Before the section is laid out, however, a value for *deltax* must be determined. *Deltax* (parameter) is calculated using the function *Calcdeltax*. This function will pass back a nonzero value for *deltax* only if additional I/O nodes have been inserted between the I/O node associated with the column presently being laid out and the I/O node associated with the adjacent column (section). *Calcdeltax* will pass back a zero value if the product term is the last product term (section) of the special structure. This is because the last reserved area section has no adjacent section associated with that structure. Consequently, the width of the last section is independent of the next I/O node. *Calcdeltax* determines the number of the product

term to the right of the product term in question. The number of I/O nodes, without product terms, between the product terms (sections) may then be determined. The number of I/O nodes is multiplied by the horizontal pitch (the horizontal pitch of I/O nodes without product terms is a fixed value) to obtain the *deltax* (parameter) value. *Deltax* is then passed to the section macro so that the appropriate rectangles may be expanded. Thus, proper section to section abutment will be maintained even though additional I/O nodes are placed between sections of a special structure.

Once the *deltax* value has been calculated, the section macro may be laid out. Each special structure has a separate layout procedure which is called by *columnlay*. *Muxlay* is the procedure used to lay out the sections of the mux structure. The parameters to *muxlay* are the reference point of the product term diffusion bar, the section counter value, the width of the product term diffusion bar (drive transistor width), and the *deltax* value. *Muxlay* may be described as follows.

```

procedure MUXLAY:
begin
  Case section counter of
  1: begin
    Set load transistor pointer to load transistor node
    in first mux section (mux1load).
    Set length of drive transistors in reserved area
    (mux1drtrlgth).
    Perform muxcell1.
  end.
  2: begin
    Set load transistor pointer to load transistor node
    in second mux section (mux2load).
    Set length of drive transistors in reserved area
    (mux2drtrlgth).
    Perform muxcell2.
  end.
  3: begin
    Set load transistor pointer to load transistor node
    in third mux section (mux3load).
    Set length of drive transistors in reserved area
    (mux3drtrlgth).
    Perform muxcell3.
  end.
  4: begin
    Set load transistor pointer to load transistor node
    in last mux section (mux4load).
    Set length of drive transistors in reserved area
    to 0 (no reserved area drive transistors).
    Perform muxcell4.
    Set section counter to 0.
  end.

```

```

end. (* case *)
end.

```

Muxlay first obtains any data required by a particular section, such as the correct device sizes, and then calls the macro procedure which lays out the appropriate section of the special structure. As described in section 4.2.2., a macro procedure may increase the *deltax* value to accommodate variations in device sizes. Any increase in *deltax* created in the macro procedure (*deltax (internal)*) must also increase the horizontal pitch between the I/O nodes (*wterm*). This increase created by *deltax (internal)* can be determined by calculating the difference between the value of *deltax* passed to the procedure (from *calcdeltax*) and the value of *deltax* passed from the procedure. The *wterm* value, along with an adjustment to the x-coordinate of the next start point and an adjustment to the position of the section I/O node, may then be adjusted accordingly. The updated values are then used to lay out all the metal bars in the cell. One additional adjustment, however, must be made. Unlike most metal bars, the GND metal-A bar is laid out prior to special structure sections. Any increase in pitch created within the macro procedure has not been added to the length of the GND bar. Therefore, a metal-A section, with a length equal to *deltax (internal)*, must be added to the GND bar to compensate for this increase.

After the transistors and special structure sections are laid out the metal bars must be laid out. Metal-B bars may be laid out in either the routable or reserved areas. The presence of a metal-B node must be tested for each row in the cell. For each row in a column, function *needmtlb* is used to determine whether a metal-B bar should be laid out. *Needmtlb* searches the metal-B linked list to determine if a metal-B bar is needed. Metal-B bars are laid out using the procedure *mtlblay*. This procedure lays out a metal-B node with a height equivalent to the vertical pitch between I/O nodes. *Mtlblay* also lays out all metal-B to metal-A contacts. The only modifications made to *mtlblay* were to the sections used to lay out contacts. A parameter was established which may be used to cancel the layout of contacts. This parameter was created because all contacts in a reserved area are hard coded as part of the section macro. The parameter (*setcontact*) must be set *true* if contacts are to be automatically laid out by GEOLAY.

Needmtla searches the metal-A linked list to determine if a metal-A bar is needed. Unlike the metal-B bars, however, metal-A bars may only be laid out in the routable area of a column. Consequently, errors due to global metal-A bars shorting with vertical metal-A bars within a reserved area will

be avoided. An error flag will be set if the user attempts to insert a metal-A bar into a reserved area. However, by not laying out any metal-A nodes in a reserved area, all undetected errors (i.e. metal-A bars created during editing connecting with vertical metal-A bars created by a section macro) will be avoided. The ability to insert transparent metal-A nodes (nodes which are present in the data structure but are not laid out in the geometric representation) into a reserved area allows the designer the option of terminating a metal-B bar anywhere within a reserved area. This option is used to insert the input and output metal-B bars of a special structure. The I/O bars must be capable of terminating, without any contact being produced, on the row containing the I/O contact in the reserved area. A transparent metal-A node may be used to represent the I/O contact allowing the metal-B bar to terminate at the contact. Then, by setting the *setcontact* parameter to *false*, a metal-B bar will terminate at the hard coded I/O contact.

As the library of special structures increases, slight modifications to GEOLAY may be unavoidable. However, with the modifications presented here, no further large scale modifications should be required. Each new special structure and special structure section will only require additions to the appropriate case statements. A procedure similar to *muxlay* will also be required to produce the actual layouts. As the number of special structure sections increases, the need for a separate procedure containing case statements may be necessary.

5.4. Symbolay

SYMBOLAY also required modifications to support special structures. These modifications are basically the same as those required by GEOLAY. Unlike GEOLAY, however, the modifications which were needed to expand the special structure sections and vary horizontal pitch (*deltax*) were not required in SYMBOLAY. The pitch between I/O nodes in the symbolic layout is independent of the structure type. Symbolic representations are not bound by the geometric design rules. The pitch, therefore, is also independent of device sizes. The symbolic representation of a special structure section is simply a rectangle outlining the reserved area. Interconnect within the reserved areas will not be diagrammed. These outlines represent the interconnect and transistors within a section of a special structure. Design of additional section macros for symbolic diagrams, consequently, may be avoided. Only I/O contacts and the global metal-B routing are shown within a reserved area outline. The width of each section is a constant (equal

to the horizontal pitch between I/O nodes) and is independent of special structure type. The height, however, is dependent upon which special structure is being represented. As in GEOLAY, the height of the reserved area is determined using the compensation height constant. By using the compensation height and horizontal pitch, the size of the reserved area section outline may be determined.

No metal-A routing is allowed into any of the section outlines. Unlike the geometric layout, all metal-A nodes in a reserved area are shown in the symbolic representation. An error message, however, will appear on the screen warning the user that a metal-A bar has been inserted into the reserved area. The user may then delete the appropriate metal-A bar. Modifying any device sizes within the reserved area of a section may be done by simply indicating MODIFY SPECIAL STRUCTURE and then using the tablet cursor to identify the appropriate section.

A label must also be associated with each special structure section. This label, placed within the section outline, will identify the section for the user. For example, the sections associated with the mux structure may be labeled mux1, mux2, mux3, and mux4 accordingly. This label may be placed in the lower left corner of the outline. Therefore, the only major modification required to SYMBOLAY was the addition of the reserved area outlines. Although the modifications required for the outlines have been completed, the insertion of section labels has yet to be implemented.

5.5. TCELL Modifications

TCELL must be able to create a special structure from a CDL function statement. The CDL function statement must identify the structure type (from the library of special structures) and identify the input and output variables. An expansion of the CDL syntax is required for defining special structures. All expansions to the syntax require modifications to the CDL scan-in procedures in TCELL. This syntax expansion is described in figure 5.3 [Luhu83]. The outputname is similar to $f[x]$ used for random logic (see figure 1.1). NOR is the functionname identifying random logic functions. Special structures, however, require different function names which identify the type of structure. For debugging purposes, the CDL functionname established should be similar to the structuretype identifier used in ICE. Input and output names are declared in the same manner as the random logic input and output names. The I/O name used in the function statement must be declared in the interconnect (TERM) section of the CDL program.

```

functionstatement = outputname "=" functionname
                  "(" inputname {";" inputname} ")"
                  {otherparameter {";" otherparameter}}
outputname = variablename
inputname = variablename | productterm
otherparameter = integer
functionname = "NOR" | "MUX" | "REG" | "DECODER" | ...

```

Figure 5.3 CDL Syntax Expansion

The "otherparameter" values may be used to manually set device sizes or delay time. These parameters will then be used in the *devsiz* procedure in TCELL. An example of an input description for a 2-1 multiplexer is as follows:

```

muxoutput = MUX (muxinputzero, muxinputone, muxselect)
               [muxdelay] ;

```

Modifications must also be made to the *Inplace*, *Pairs*, and *Rout* procedures in TCELL. *Inplace* must contain a procedure similar to the *insertmux* procedure used in ICE. This procedure must create the product terms required for the special structure along with establishing the proper structure types. As in TCELL, each special structure may require a separate procedure. An update to the procedure used to create the random logic gates is also required. The *structuretype* identifier must be set to "nosstr" for each random logic product term. *Pairs* must then be modified so that each special structure is treated as one function rather than a set of individual product terms. This will assure that ordering of the special structure product terms remains consistent and that no random logic product terms are placed between sections of the structure. *Rout* must then be modified to automatically rout the special structure I/O bars and the select bar. Presently, these modifications have not been incorporated into TCELL. Portions of TCELL are still in the debugging stage and require special modifications. Once these modifications approach completion, the modifications for special structures may be made.

5.6. Summary

Five basic steps are required to add a special structure to the system library. These steps are: create menu entry for new structure, declare a node for each section of the structure (ddefstype), add appropriate insert and delete (and modify if required) procedure(s) into ICE and TCELL, update all structure type "case" statements in ICE, TCELL, GEOLAY, and SYMBOLAY, and declare height and width constants for each section (ddefcons.i).

An addition which might be desirable would be to extract all of the "case" statements (dependent upon structure type) and declare the statements as individual procedures. These procedures may then be placed in a separate file. New structures may then be added by simply updating the procedures in this one file. This will also eliminate the possibility of producing unintentional changes to either of the source files. The insert, delete, and modify procedures may also be declared in a separate file which will completely eliminate the need to edit ICE or TCELL. Presently ICE has been divided according to procedures (/mntb/1/kubitz/work/midas/src/ice) but the "case" statements have not been extracted. The procedures required for a new structure may simply be declared in a file and added to this directory. The makefile will then require updating to include this new file.

CHAPTER 6

CONCLUSION

Special structures were introduced into the system to minimize the area requirements for a number of commonly used functions such as multiplexers, flipflops, and registers. Area reductions were created through the use of manually designed macros which used interconnect not normally allowed by the data structure. The data structure allows only vertical metal-A routing whereas a special structure macro may use vertical and/or horizontal metal-A along with polysilicon routing. The drawback, however, is the limited amount of global routing allowed through a special structure (no metal-A global routing) and the variations in horizontal pitch required by various sections of a structure as compared to the fixed pitch of a random logic function. These variations in pitch may require a number of recursive calculations to determine the pitch of adjacent cells. The pitch of the adjacent cells must be varied accordingly to allow these cells to properly abut along the top and bottom edges of the cell containing the special structure.

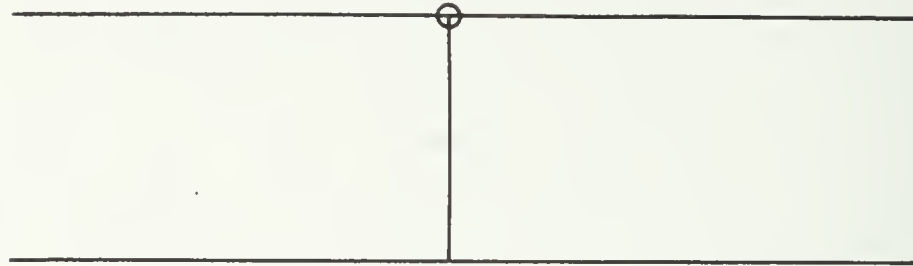
The option of using hard coded designs in cell layouts greatly expands the capabilities of the layout synthesis system. In addition to reducing overall chip area, hard coded designs may be used to expand the logic capabilities of the system. The incorporation of dynamic logic cells is one example of the possible use of hard coded designs. In the case of structures such as register cells, dynamic logic could significantly reduce area requirements. Dynamic logic structures would, however, require a few modifications to the data structure because some sections of a dynamic logic structure may not contain a product term. This special structure can not then be associated with any product term. Thus, slight modifications may be required to handle these cases.

Incorporating transmission gates into the data structure is also an option which may be considered. A transmission gate may simply be considered a special structure which does not contain a load transistor. No reserved area would be required for a transmission gate. Unlike present special structures, a transmission gate may not be treated simply as a special case of a product term. If this were the case, the reserved area would have to be the entire height of the cell, and thus no vertical diffusion track could be laid out. A new node is required in the data structure and modifications to TCELL, ICE, GEOLAY, and SYMBOLAY would be needed to incorporate transmission gates. Presently, modifications have been

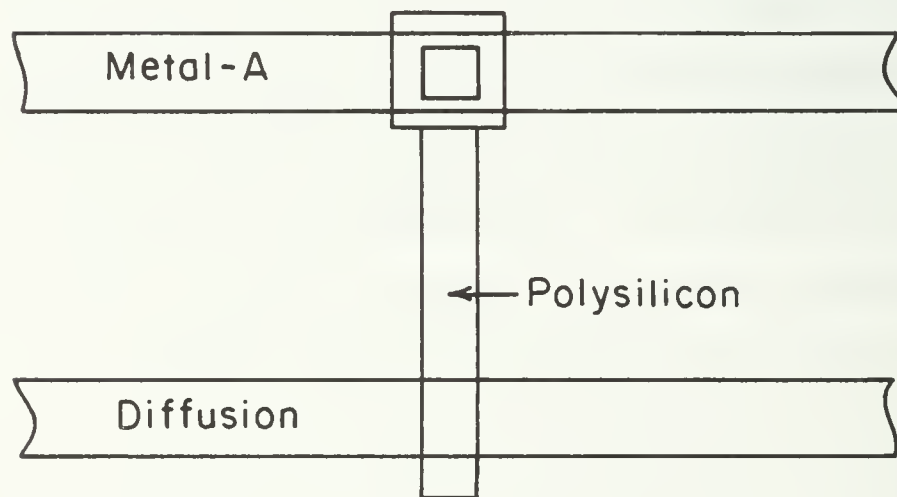
made only for special structures which have product terms associated with them. Symbolically, a transmission gate could be represented as a drive transistor along a horizontal diffusion path or have an entirely unique representation, such as a circle, which would be placed along the horizontal diffusion path. This diffusion path must be connected between two functions. If the two product terms associated with the functions are not adjacent, a provision must be made so that the transmission gate diffusion path may connect to a metal-A bar. The metal-A bar must then be capable of connecting to a product term. Presently the data structure does not allow a diffusion to metal-A contact except along the VDD and GND bars. A solution would be to insert metal-A to diffusion contacts (metal-A to metal-B contacts may also be desirable) into the data structure. Once the diffusion path is connected to two product terms, a pass transistor must be inserted along the diffusion. This may be done by using the drive transistor macro `drtrans.i`. This macro allows polysilicon to extend from either side of the metal-A to polysilicon contact. If the contact is to connect to a metal-B bar, a section of metal-A must be extended to the appropriate column. An example of a transmission gate is shown in figure 6.1. The parameters to a transmission gate would be the row number of the diffusion path, the row number of the pass transistor contact, the end points of the diffusion path, the end points of the metal-A bar connected to the pass transistor, and the width and length of the pass transistor. A macro may then lay out the diffusion path and the transistor. GEOLAY and SYMBOLAY will be capable of laying out the metal bars as long as all metal-A contacts associated with the structure have been inserted into the metal-A linked list.

Presently, only the modifications required for NMOS special structures have been implemented. The modifications described in this thesis should, however, give the designer a good indication of the changes needed to implement any new special structure. Possibly, the greatest difficulty will arise in attempting to implement new types of structures into CDL. Transmission gates, for example, may be difficult to implement into a CDL description.

A library of special structures should be created which will present the user with a description of each of the structures presently available. This source should provide the information needed for the CDL description along with possibly a rough estimate of delay time and power requirements for a given set of device sizes. With this information a user will be able to insert any structure into the CDL description of a cell. The possibility of allowing a designer to create his/her own special structures might also be



Symbolic



Geometric

Figure 6.1 Transmission Gate

desirable. This possible option, along with the use of different dynamic logic functions, will significantly expand the capabilities of this system. The implementation of NMOS special structures alone, however, is a very useful first step in reducing the overall area required by cells created through this automatic layout synthesis system.

APPENDIX A

MUX PROCEDURES

A listing of the code used to lay out the geometric representation of the 2-1 multiplexer is presented in this appendix. Each procedure (muxcell1, muxcell2, ...) represents one section of the cell.

Parameters:

refpoint: The upper-left-most point of the restricted area.
 Uses the same Y-coordinate as would a metal-A bar.

deltax: An integer used when the distance between the
 diffusion bars needs to be increased.

drtranswidth: The width of the diffusion bar between
 transistors. (1, 2, or 3 times wdiff).

drtranslength: The length of the drive transistors.
 ldtranswidth: The width of the diffusion bar
 used in the load transistor.

ldtranslength: The length of the diffusion bar used in the
 load transistor.

orientation: The orientation of the cell.

```

procedure muxcell1 (var refpoint : coordinate ;
                    var deltax : integer ;
                    drtranswidth : integer ;
                    drtranslength : integer ;
                    ldtranswidth : integer ;
                    ldtranslength : integer ;
                    orientation : integer) ;

var cornera, corneraa, cornerb, cornerc, cornerd, cornere,
    cornerf, cornerg, cornerh : coordinate ;

widthb, widthc, widthd, heighte, widthf, widthg : integer ;
dmcontlength, pmcontlength, diffheight : integer ;
deltax2, totalwidth, totalheight : integer ;

begin {MUX1}
  refpoint.x := refpoint.x - mux1width + mux1db + wdiff ;
  pmcontlength := (2 * pmcutclear) + wfirstcut ;
  dmcontlength := (2 * dmcutclear) + wfirstcut ;
  deltax2 := 0 ;
  diffheight := pmcutclear + wmtla + (3 * mtlbmtlb) ;

```

```

{diffheight(min.) := pmcutclear + wmtla + mtlbmtlb ;}

(* These rectangles may be used to test reference points *)
(* and constant values *)
{cornera.x := refpoint.x + 30 ;
cornera.y := refpoint.y + wmtla ;
maskbox (cornera, wmtla, diffpoly, metala, orientation) ;
cornera.x := refpoint.x + 30 - 14 ;
maskbox (cornera, wmtla, diffpoly, polysilicon, orientation) ;
maskbox (refpoint, diffpoly, diffpoly, polysilicon, orientation) ;}

cornera.x := refpoint.x + polygateclear + dmcutclear + mtlamtla ;
cornera.y := refpoint.y ;
loadtrans (cornera, drtranswidth, diffheight, ldtranswidth,
ldtranslength, orientation) ;
cornera.y := cornera.y - mtlbmtlb - wpoly ;
cornerb.x := cornera.x + drtranswidth + diffpoly ;
cornerb.y := cornera.y ;
if ldtranslength > (2 * diffpoly)
    then deltax2 := ldtranslength - (2 * diffpoly) ;
if deltax2 > deltax
    then deltax := deltax2 ;
widthb := diffpoly + drtranswidth + polygateclear ;
if drtranswidth = (3 * wdif)
    then widthb := widthb + wdif ;
drtrans (cornerb, 2, wfirstcut, drtranslength, widthb, orientation) ;
cornerc.x := cornerb.x + pmcutclear + wfirstcut ;
cornerc.y := cornerb.y - pmcutclear ;
widthc := wpoly + pmcutclear + deltax ;
maskbox (cornerc, wmtla, widthc, metala, orientation) ;
cornerd.x := cornera.x - polygateclear ;
cornerd.y := cornerb.y - (2 * polypoly) - wpoly ;
widthd := polygateclear + (2 * wdif) + diffpoly + pmcontlength
+ deltax ;
maskbox (cornerd, drtranslength, widthd, polysilicon, orientation) ;
cornere.x := cornerd.x + widthd - wpoly ;
cornere.y := cornerd.y - wpoly ;
heighte := (2 * diffpoly) + (3 * wdif) + wpoly ;
maskbox (cornere, heighte, wpoly, polysilicon, orientation) ;
cornerf.x := cornere.x ;
cornerf.y := cornere.y - heighte + wpoly ;
widthf := wpoly ;
maskbox (cornerf, wpoly, widthf, polysilicon, orientation) ;
totalwidth := dmcutclear + (3 * wdif) + diffpoly + pmcontlength
+ wmtla + deltax ;
totalheight := diffheight + wdif + dmcontlength + dmcutclear
+ diffpoly + ldtranswidth + (2 * wdif) + dmcontlength ;
cornerg.x := cornera.x + wdif ;
cornerg.y := cornerf.y + (2 * wfirstcut) ;
widthg := diffpoly + deltax2 ;
maskbox (cornerg, dmcontlength, widthg, metala, orientation) ;
cornerh.x := cornerg.x + widthg - dmcutclear ;
cornerh.y := cornerg.y + dmcutclear ;
contacts (3, cornerh, wsecndcut, wsecndcut, orientation) ;

```

```

    (* Used to test new reference point *)
    refpoint.x := refpoint.x + totalwidth ;
    refpoint.y := refpoint.y ;
    {maskbox (refpoint, wmtla, diffpoly, diffusion, orientation) ;}
end ; {MUX1}

procedure muxcell2 (var refpoint : coordinate ;
                    var deltax : integer ;
                      drtranswidth : integer ;
                      drtranslength : integer ;
                      ldtranswidth : integer ;
                      ldtranslength : integer ;
                      orientation : integer) ;

var cornera, cornerb, cornerc, cornerd, cornere, cornerf,
    cornerg, cornerh, corneri : coordinate ;

widthb, widthc, heightd, widthg : integer ;
dmcontlength, pmcontlength, diffheight : integer ;
deltax2, totalwidth, totalheight, boundarydiff : integer ;

begin {MUX2}
    deltax2 := 0 ;
    refpoint.x := refpoint.x - mux2width + mux2db + wdiff ;
    pmcontlength := (2 * pmcutclear) + wfirstcut ;
    dmcontlength := (2 * dmcutclear) + wfirstcut ;
    diffheight := pmcutclear + wmtla + (3 * mtlbmtlb) ;
    {diffheight(min.) := pmcutclear + wmtla + mtlbmtlb ;}

    (* Used to test values for totalwidth and diffusion to boundary *)
    (* constants *)
    {cornera.x := refpoint.x + 28 ;
    cornera.y := refpoint.y + wmtla ;
    maskbox (cornera, wmtla, diffpoly, metala, orientation) ;
    cornera.x := refpoint.x + 28 - 16 ;
    maskbox (cornera, wmtla, diffpoly, polysilicon, orientation) ;
    maskbox (refpoint, diffpoly, diffpoly, polysilicon, orientation) ;}

    boundarydiff := 28 - 16 - wdiff ;
    cornera.x := refpoint.x + boundarydiff ;
    cornera.y := refpoint.y ;
    loadtrans (cornera, drtranswidth, diffheight, ldtranswidth,
    ldtranslength, orientation) ;
    cornera.y := cornera.y - mtlbmtlb - wpoly ;
    if drtranswidth = (3 * wdiff)
        then deltax2 := wdiff ;
    if ldtranslength > ((6 * diffpoly) + deltax2)
        then deltax2 := deltax2 + ldtranslength - (6 * diffpoly) ;
    if deltax2 > deltax
        then deltax := deltax2 ;
    cornerb.x := refpoint.x ;

```

```

cornerb.y := cornera.y - wpoly - (3 * wdiff) - (2 * pmcontlength) ;
widthb := diffpoly ;
maskbox (cornerb, wpoly, widthb, polysilicon, orientation) ;
cornerc.x := cornera.x - polygateclear ;
cornerc.y := cornera.y - wpoly + round(drtranslength/2) ;
widthc := polygateclear + (3 * wdiff) ;
maskbox (cornerc, drtranslength, widthc, polysilicon, orientation) ;
cornerd.x := cornerc.x + widthc - diffpoly ;
cornerd.y := cornerc.y ;
heightd := (2 * wmtla) ;
maskbox (cornerd, heightd, wpoly, polysilicon, orientation) ;
cornere.x := cornerd.x ;
cornere.y := cornerd.y - heightd ;
contacts (2, cornere, wfirstcut, wfirstcut, orientation) ;
maskbox (cornere, diffpoly, pmcontlength, metala, orientation) ;
cornerf.x := cornere.x - pmcutclear ;
cornerf.y := cornere.y ;
contacts (3, cornerf, wsecndcut, wsecndcut, orientation) ;
cornerg.x := retpoint.x ;
cornerg.y := cornera.y - pmcutclear ;
totalwidth := boundarydiff + (3 * wdiff) + diffpoly + wmtla + deltax ;
widthg := totalwidth ;
maskbox (cornerg, wmtla, widthg, metala, orientation) ;
totalheight := diffheight + wdiff + dmcontlength + (2 * dmcutclear)
+ ldtranswidth + (2 * wdiff) + dmcontlength ;

(* Used to test new reference point *)
retpoint.x := retpoint.x + totalwidth ;
retpoint.y := retpoint.y ;
{maskbox (retpoint, wmtla, diffpoly, diffusion, orientation) ;}
end ; {MUX2}

```

```

procedure muxcell3 (var retpoint : coordinate ;
                    var deltax : integer ;
                    drtranswidth : integer ;
                    drtranslength : integer ;
                    ldtranswidth : integer ;
                    ldtranslength : integer ;
                    orientation : integer) ;

var cornera, cornerb, cornerc, cornerd, cornere, cornerf,
    cornerg, cornerh, corneri : coordinate ;

```

```

widthb, heightc, widthd, widthe, heightg, widthi : integer ;
dmcontlength, pmcontlength, diffheight, deltax2 : integer ;
totalwidth, totalheight, boundarydiff : integer ;

```

```

begin {MUX3}
    retpoint.x := retpoint.x - mux3width + mux3db + wdiff ;
    deltax2 := 0 ;
    pmcontlength := (2 * pmcutclear) + wfirstcut ;

```

```

dmcontlength := (2 * dmcutclear) + wfirstcut ;
diffheight := pmcutclear + wmtla + (3 * mtlbmtlb) ;
{diffheight(min.) := pmcutclear + wmtla + mtlbmtlb ;}

{cornera.x := respoint.x + 34 ;
cornera.y := respoint.y + wmtla ;
maskbox (cornera, wmtla, diffpoly, metala, orientation) ;
cornera.x := respoint.x + 34 - 14 ;
maskbox (cornera, wmtla, diffpoly, polysilicon, orientation) ;
maskbox (respoint, diffpoly, diffpoly, polysilicon, orientation) ;}

boundarydiff := 34 - 14 - wdif ;
cornera.x := respoint.x + boundarydiff ;
cornera.y := respoint.y ;
loadtrans (cornera, drtranswidth, diffheight, ldtranswidth,
ldtranslength, orientation) ;
cornera.y := cornera.y - mtlbmtlb - wpoly ;
if drtranswidth = 3 * wdif
    then    deltax := deltax + wdif ;
if ldtranslength > (3 * diffpoly)
    then deltax2 := ldtranslength - (3 * diffpoly) ;
if deltax2 > deltax
    then deltax := deltax2 ;
totalwidth := dmcutclear + wpoly + (2 * polypoly) +
polygateclear + polypoly + pmcontlength
+ wpoly + deltax ;
cornerb.x := respoint.x ;
cornerb.y := cornera.y ;
contacts (2, cornerb, wfirstcut, wfirstcut, orientation) ;
cornerc.x := cornerb.x + pmcutclear ;
cornerc.y := cornerb.y ;
heightc := mtlbmtlb + wdif + pmcontlength + (2 * wpoly) ;
maskbox (cornerc, heightc, wpoly, polysilicon, orientation) ;
cornerd.x := cornerc.x + wpoly ;
cornerd.y := cornerc.y - heightc + wpoly ;
widthd := polypoly ;
maskbox (cornerd, wpoly, widthd, polysilicon, orientation) ;
cornere.x := cornera.x - polygateclear ;
cornere.y := cornera.y - wpoly + round(drtranslength/2) ;
widthe := polygateclear + (2 * wdif) + deltax2 ;
if (drtranswidth = (3 * wdif)) and (deltax2 < wdif)
    then widthe := widthe + wdif - deltax2 ;
if (drtranswidth = (2 * wdif)) and (deltax2 < diffpoly)
    then widthe := widthe + diffpoly - deltax2 ;
maskbox (cornere, drtranslength, widthe, polysilicon, orientation) ;
cornerf.x := cornere.x + widthe ;
cornerf.y := cornera.y ;
contacts (2, cornerf, wfirstcut, wfirstcut, orientation) ;
cornerg.x := cornerf.x + pmcutclear ;
cornerg.y := cornerf.y ;
heightg := diffpoly + (2 * wpoly) + wmtla + wdif ;
maskbox (cornerg, heightg, wmtla, metala, orientation) ;
cornerh.x := cornerg.x - pmcutclear ;
cornerh.y := cornerg.y - heightg + wpoly ;
contacts (3, cornerh, wsecndcut, wsecndcut, orientation) ;

```

```

corneri.x := cornera.x - polygateclear ;
corneri.y := cornera.y - (2 * pmcutclear) - wpoly - polypoly ;
widthi := totalwidth - polypoly - pmcontlength ;
maskbox (corneri, drtranslength, widthi, polysilicon, orientation) ;
totalheight := diffheight + wdif + dmcontlength + (2 * dmcutclear)
+ ldtranswidth + (2 * wdif) + dmcontlength ;

refpoint.x := refpoint.x + totalwidth ;
refpoint.y := refpoint.y ;
{maskbox (refpoint, wmtla, diffpoly, diffusion, orientation) ;}
end ; {MUX3}

```

```

procedure muxcell4 (var refpoint : coordinate ;
                    var deltax : integer ;
                    drtranswidth : integer ;
                    drtranslength : integer ;
                    ldtranswidth : integer ;
                    ldtranslength : integer ;
                    orientation : integer) ;

```

```

var cornera, cornera3, cornerc, cornerd, cornere, cornerb,
    cornerf, cornerg : coordinate ;

```

```

heightb, widthc : integer ;
dmcontlength, pmcontlength, diffheight : integer ;
totalwidth, totalheight, boundarydiff, diffboundary : integer ;

```

```

begin {MUX4}
    refpoint.x := refpoint.x - mux4width + mux4db + wdif ;
    pmcontlength := (2 * pmcutclear) + wfirstcut ;
    dmcontlength := (2 * dmcutclear) + wfirstcut ;
    diffheight := pmcutclear + wmtla + (3 * mtlbmtlb) ;
    {diffheight(min.) := pmcutclear + wmtla + mtlbmtlb ;}

    {cornera.x := refpoint.x + 30;
    cornera.y := refpoint.y + wmtla ;
    maskbox (cornera, wmtla, diffpoly, metala, orientation) ;
    cornera.x := refpoint.x + 30 - 14 ;
    maskbox (cornera, wmtla, diffpoly, polysilicon, orientation) ;
    maskbox (refpoint, diffpoly, diffpoly, polysilicon, orientation) ;}

    boundarydiff := 30 - 14 - wdif ;
    cornera.x := refpoint.x + boundarydiff ;
    cornera.y := refpoint.y ;
    loadtrans (cornera, drtranswidth, diffheight, ldtranswidth,
    ldtranslength, orientation) ;
    cornera.y := cornera.y - mtlbmtlb - wpoly ;
    if ldtranslength > (wmtlb + deltax)
        then deltax := deltax + ldtranslength - wmtlb ;
    totalwidth := wpoly + (3 * wdif) + diffpoly + deltax ;
    cornerb.x := refpoint.x ;

```

```

cornerb.y := cornera.y - pmcontlength - polypoly ;
heightb := wdiff + dmcontlength + (3 * wpoly) ;
maskbox (cornerb, heightb, wpoly, polysilicon, orientation) ;
cornerc.x := cornerb.x + wpoly ;
cornerc.y := cornerb.y - heightb + wpoly ;
widthc := diffpoly ;
maskbox (cornerc, wpoly, widthc, polysilicon, orientation) ;
totalheight := diffheight + wdiff + dmcontlength + (2 * dmcutclear)
+ ldtranswidth + (2 * wdiff) + dmcontlength ;

refpoint.x := refpoint.x + totalwidth ;
refpoint.y := refpoint.y ;
{maskbox (refpoint, wmtla, diffpoly, diffusion, orientation) ;}
end ; {MUX4}

```

APPENDIX B

SECTION NODES

A listing of the four multiplexer section nodes, as they appear in ddeftype.i, is given in this appendix. Also shown are the other type declarations in ddeftype.i which must be updated for each new special structure.

type

The linked lists

```
funcptr    = ^funclist;
funclist = record
    nextfunc : funcptr;
    funcdata : funct
end;

nodetype = (
    headers,                (* headers to the linked lists *)
    varinode,               (* logic variables *)
    ionode,                 (* i/o node on cell's edges *)
    prodtermnode,           (* product-term, diffusion bar *)
    drivtransnode,          (* drive transistor *)
    loadtransnode,          (* load transistor *)
    mtlanode,               (* first-metal bar *)
    mtlbnode,               (* second-metal bar *)
    mux1node,               (* first sect. of mux *)
    mux2node,               (* second section of mux *)
    mux3node,               (* third section of mux *)
    mux4node);              (* fourth section of mux *)

structuretype = (
    mux,                    (* 2-1 multiplexer *)
    nosstr);                (* no special structure *)
                          (* i.e. random logic *)

nodeptr = ^noderecord;
noderecord = record
    nextnode : nodeptr;
    case tag : nodetype of
        prodtermnode : (
            ptflag,          (* 1 if leftmost p-t *)
            ptnumber :      (* number, left-to-right *)
                integer;
            ptss,            (* to special structure node *)
            ptdrtrans,       (* to drive-trans list *)
```

```

    ptleft,          (* to left p-t          *)
    ptright,         (* to right p-t         *)
    ptmtla,          (* to first-metal bar   *)
    ptload :         (* to load transistor    *)
                      nodeptr;

    ptstrtype :      (* to structure type    *)
                      structuretype);

mux1node : (
    mux1drtrwidth,   (* drive trans. width    *)
    mux1drtrlgth :   (* drive trans. length    *)
                      integer ;

    mux1load,        (* to load transistor     *)
    muxoutmtlb :     (* to second-metal bar    *)
                      nodeptr);

mux2node : (
    mux2drtrwidth,   (* drive trans. width    *)
    mux2drtrlgth :   (* drive trans. length    *)
                      integer ;

    mux2load,        (* to load transistor     *)
    muxin0mtlb :     (* to logic 0 input bar   *)
                      nodeptr);

mux3node : (
    mux3drtrwidth,   (* drive trans. width    *)
    mux3drtrlgth :   (* drive trans. length    *)
                      integer ;

    mux3load,        (* to load transistor     *)
    muxin1mtlb :     (* to logic 1 bar        *)
                      nodeptr);

mux4node : (
    mux4load :       (* to load transistor     *)
                      nodeptr);

end;

```

REFERENCES

- [Bilg82] Bilgory, A., "Compilation of Register Transfer Language Descriptions Into Silicon," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, Report UIUCDCS-R-82-1091, 1982.
- [Esak83] Esakov, J., "Multi-Level Interactive Design Automation Supervisor," M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Report UIUCDCS-R-83-1137, 1983.
- [Gajs82] Gajski, D.D., "The Structure of a Silicon Compiler," Int. Conf. on Circuits & Computers, 1982.
- [Luhu83] Luhukay, J.F.P., "Layout Synthesis of NMOS Gate-Cells," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1983.
- [MeCo80] Mead, C. and Conway, L., "*Introduction to VLSI Systems*," Reading, MA: Addison-Wesley, 1980.
- [MoSC79] Morrissey, T., Schroeder, L., and Courtney, J., "Detailed Design Document for the Pascal Plotting Facility," unpublished but available as a user's manual from the University of Illinois at Urbana-Champaign, Department of Computer Science.
- [Murog82] Muroga, S., "*VLSI System Design*," New York, NY: Wiley & Sons, 1982.
- [Unix79] "Unix Programmer's Manual," 7th Edition, Virtual VAX-11 Version, Department of Electrical Engineering and Computer Science, University of California Berkeley, Dec. 1979.
- [Wong82] Wong, F.W., "Interactive Cell Editor, A Symbolic Editor for NMOS Gate Cells," M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Report 82-1104, 1982.
- [Yip84] Yip, P.K., M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, In Preparation.

UNIVERSITY OF ILLINOIS-URBANA



3 0112 121951138